DXNN Application (DX-APP) User Manual

v2.0.0

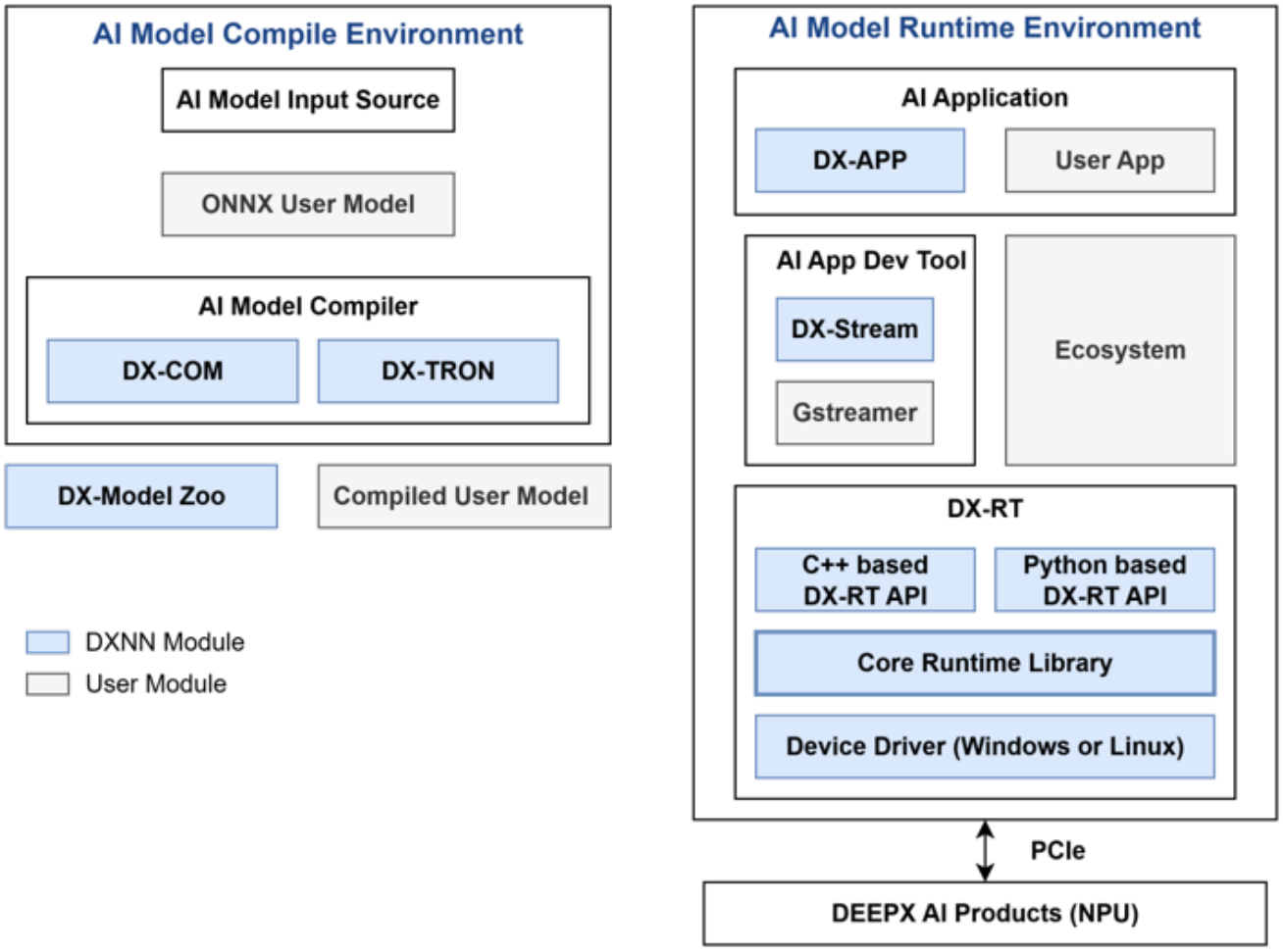
DEEPX.ai

Table of contents

1. DXNN Application Overview

This chapter provides an overview of the DEEPX SDK architecture and explains each core component, and describes the overview and key features of DX-APP.

1.1 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 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.

1.2 DX-APP Features

DX-APP is a set of application templates designed to demonstrate the performance and deployment of AI models on DEEPX NPUs. It provides ready-to-use examples for image classification, object detection, segmentation, and pose estimation.

You can quickly evaluate inference capabilities without modifying the source code and easily adapt the templates for custom applications. These templates significantly reduce the overhead of environment configuration and manual implementation.

Note. Application performance may vary depending on host system specifications. Each demo includes pre-processing, post-processing, and graphics processing operations.

1.2.1 Demos

DX-APP demos are optimized to showcase pre-compiled models on DEEPX NPUs with minimal setup. Each demo represents a common AI task and can be executed using images, videos, or live camera input.

Classification

- Executes classification models with image inputs (e.g., `224x224`).
- Outputs the Top-1 predicted class.
- Example: `example/run_classifier/imagenet_example.json`

Object Detection

This demo supports image, video (mp4, mov, avi), and camera input.

- For image input, outputs result.jpg and prints detected objects to the terminal.
- For video input, displays bounding boxes on the output video.

Pose Estimation

- Detects people and estimates keypoints (joints) using image, video, or camera input.
- The output includes both bounding boxes and joint coordinates rendered on screen.

Segmentation

This demo uses two models to perform segmentation.

- For image input, saves results to result.jpg and prints info to the terminal.
- For video input, displays output with both detection boxes and segmentation masks.

1.2.2 Templates

DX-APP provides lightweight Application Templates that run classification or detection models by modifying only the JSON configuration. No code changes required.

Classification

- Supports various classification models with image or binary input.
- Outputs the Top-1 class by adjusting JSON fields.
- Example JSON: `example/run_classifier/imagenet_example.json`

Object Detection

- Supports YOLO-series models using image, binary, video, RTSP stream, or camera input.
- Built on multi-threading for multi-channel and real-time processing.
- Supports dynamic input source expansion and grid-style output display.
- Example JSON: `example/run_detector/yolov5s3_example.json`

---

2. DX-APP Installation and Build

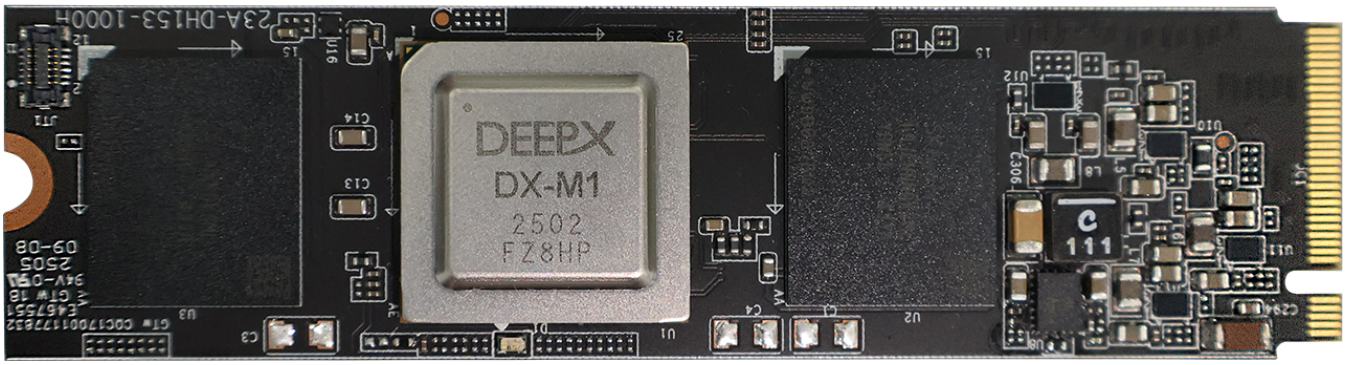
This chapter describes the system requirements and the installation instructions on Linux and WIndows to use **DX-APP**.

2.1 System Requirements

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

Hardware Requirements

- **CPU:** amd64(x86_64), aarch64(arm64)
- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space
- The system **must** support connection to an **M1 M.2** module with the M.2 interface on the host PC.



Note. The **NPU Device Driver** and **DX-RT Library must** be installed. Refer to **DX-RT User Manual** for step-by-step installation instructions.

2.2 Installation on Linux

This section describes the software requirements and installation steps for setting up **DX-APP** on Ubuntu-based systems.

2.2.1 Software Requirements on Linux

To run **DX-APP** on Linux, the following components **must** be installed.

- **OS**: Ubuntu 18.04 / 20.04 / 22.04 / 24.04 (x64)
- **Deepx M1 Driver Version**: v1.7.1 or higher
- **Deepx M1 Runtime Lib Version**: v3.0.0 or higher

All required components are included in the **DXNN All Suite (DX-AS)** package.

2.2.2 Prerequisites Setup

1. Install DX-RT Device Driver
To set up the build Environment, refer to **Section. Linux Device Driver Installation** in **DX-RT User Manual**.

Once the DX-RT device driver is installed, the system should include both the PCIe driver and the runtime driver. You can verify the installation by checking the loaded kernel modules.

```
lsmod | grep dx

# dxrt_driver 53248 2
# dx_dma 475136 7 dxrt_driver
```

2. Install DX-RT Library
To install the DX-RT library and NPU device driver, refer to **Section. Build Guide for Cross-compile** in **SDX-RT User Manual**.

Once **DX-RT** is built, the runtime library and header files are installed in the following directory.

- Libraries: `/usr/local/lib`
- Headers: `/usr/local/include`

```
set(DXRT_INSTALLED_DIR /usr/local)
```

If necessary, you can modify the installation path by editing `cmake/toolchain.x86_64.cmake`.

2.2.3 DX-APP Application Setup

1. DX-APP Installation Options
You can check the available **DX-APP** installation options by running the following command.

```
./install.sh # --help
```

You can view more installation options by entering the `--help` flag.

2. OpenCV Installation Options
If you want to enable CPU/GPU acceleration, OpenCV **must** be manually installed on your system. During the OpenCV build process, setting the following flags are needed.

- `TBB=ON, IPP=ON, CUDA=ON`

If OpenCV is already installed, manually set the `OpenCV_DIR` path in your toolchain file.

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR x86_64)
set(DXRT_INSTALLED_DIR /usr/local)
set(OpenCV_DIR /your/opencv/installation/dir)
set(onnxruntime_LIB_DIRS /usr/local/lib)
```

3. Build and Run DX-APP
To build `dx_app`, run the following command.

```
./build.sh ## Use --clean for a clean build
```

To download required models and sample videos, run the following command.

```
./setup.sh
```

Assets are downloaded and placed in the `assets/` directory. The available assets include models for Classification, Object Detection, and Segmentation.

To test `dx_app`, run the following command.

```
./scripts/run_detector.sh
```

You can also use the `run_demo.sh` script to conveniently run a variety of demo applications included with DX-APP. This script provides an interactive menu that allows you to quickly test different AI models and features without needing to remember or type out complex command-line arguments.

To use the demo launcher, simply execute the following command in your terminal:

```
./run_demo.sh
  0: Object Detection (YOLOv7)
```

```
  1: Object Detection (YOLOv8N)
  2: Object Detection (YOLOv9S)
  3: Face Detection (YOLOV5S_Face)
  4: Pose Estimation
  5: Semantic Segmentation
  6: Multi-Channel Object Detection (YOLOv5)
  7: Multi-Model Object Detection (YOLOv5) & Segmentation
which AI demo do you want to run:(timeout:10s, default:0)
```

4. Resolve Shared Library Errors

If you encounter shared library errors (e.g., `libdxrt.so`), update the system's library cache.

```
# Copy your library to /usr/local/lib
sudo cp your_library.so /usr/local/lib

# Update the system's library cache
sudo ldconfig
```

3. Demo Guide

This chapter introduces a quick-start experience using pre-built demo applications provided by **DX-APP**. These applications allow developers to evaluate DeepX NPU performance on common vision AI tasks such as classification, detection, and segmentation. Developers can modify the examples or use them as templates to build custom applications.

Note. Performance results may vary depending on host system specifications, as the demos include host-side pre-processing, post-processing, and rendering operations.

3.1 C++ Demo Application List

DX-APP provides the following C++ demo applications.

- **Classification**: Basic Classification, ImageNet Classification
- **Object Detection**: Yolo Object Detection, Yolo Object Detection - Multi Channel
- **Pose Estimation**: Human Pose Estimation
- **Segmentation**: Semantic Segmentation DeepLabV3 (CityScape dataset Only), Semantic Segmentation DeepLabV3 (CityScape dataset Only) + Yolo Object Detection

3.2 Running Demo Executables

Each demo can be executed on Linux or Windows.

3.2.1 Classification

This section demonstrates how to run the ImageNet Classification demo on Windows.

- Model: ImageNet-trained `EfficientNetB0_4.dxnn` model
- Input: Single image file
- Output: Top-1 classification result printed to the terminal

Classification Demo

• Linux Command

```
./bin/classification -m assets/models/EfficientNetB0_4.dxnn -i sample/ILSVRC2012/1.jpeg
```

• Windows Command

```
bin\classification.exe -m assets/models/EfficientNetB0_4.dxnn -i sample/
ILSVRC2012/1.jpeg
```

• Output Example

```
Top1 Result : class 321
[DXAPP] [INFO] total time : 5703 us
[DXAPP] [INFO] per frame time : 5703 us
[DXAPP] [INFO] fps : 200
```

ImageNet Classification Demo

• Linux Command

```
./bin/imagenet_classification -m assets/models/EfficientNetB0_4.dxnn -i example/
imagenet_classification/imagenet_val_map.txt -p sample/ILSVRC2012/
```

• Windows Command

```
bin\imagenet_classification.exe -m assets\models\EfficientNetB0_4.dxnn -i
example\imagenet_classification\imagenet_val_map.txt -p sample\ILSVRC2012\
```

• Output Example

The output shows the accuracy of the classification result is **74.3%** and the frame rate (fps) is **634**.

3.2.2 Object Detection

This section explains how to run object detection demos using YOLOv5 models. Both single-stream and multi-stream inference scenarios are supported.

- Model: `yolov5s_512`

YOLO Object Detection - Single Channel
This demo performs object detection on a single input stream using a YOLOv5 model.

- Output Example: Upon execution, the application displays detected objects.

```
./bin/yolo -m assets/models/YOLOV5S_3.dxnn -i sample/1.jpg -p 1
...
  Detected 10 boxes.
    BBOX:person(0) 0.877039, (308.116, 138.729, 400.442, 363.999)
    BBOX:bowl(45) 0.760544, (25.5619, 359.37, 78.8973, 393.056)
    BBOX:bowl(45) 0.749591, (46.0043, 315.064, 107.362, 346.737)
    BBOX:oven(69) 0.706145, (0.168434, 228.332, 154.418, 324.19)
    BBOX:person(0) 0.631538, (0.423515, 295.068, 47.6304, 332.823)
    BBOX:bowl(45) 0.586059, (0.148106, 329.093, 68.946, 379.93)
    BBOX:oven(69) 0.571996, (389.631, 245.565, 495.666, 359.34)
    BBOX:bottle(39) 0.455668, (172.258, 268.919, 200.505, 322.398)
    BBOX:pottedplant(58) 0.442108, (0.457752, 86.3962, 51.3189, 208.127)
    BBOX:bowl(45) 0.407671, (124.369, 219.818, 145.423, 232.568)
  Result saved to result.jpg
  [DXAPP] [INFO] total time : 23618 us
  [DXAPP] [INFO] per frame time : 23618 us
  [DXAPP] [INFO] fps : 43.4783
```

In this example, a person is detected with **confidence 0.877**, and the bounding box is defined by the four coordinates.

Pre-processing and Post-processing Parameters

YOLO models in **DX-APP** require external configuration for pre-processing and post-processing parameters. These parameters are not embedded in the `.dxnn` model.file.

Pre-processing and post-processing parameters

- Defined in: `demos/demo_utils/yolo_cfg.cpp`
- Referenced in: `yolo_1channel.cpp`

To customize,

- Modify the existing parameters in both files, or
- Add new parameter entries with a new model name to both files.

```
YoloParam yolov5s_512 = {
    512,  // height
    512,  // width
    0.25, // confThreshold
    0.3,  // scoreThreshold
    0.4,  // iouThreshold
    0,   // numBoxes
    80,   // numClasses
    "output", // onnx output name
    {     // if use_ort = off, layers config
        createYoloLayerParam("378", 40, 40, 3, { 10.0f, 16.0f, 33.0f }, { 13.0f, 30.0f,
23.0f }, { 0 }),
```

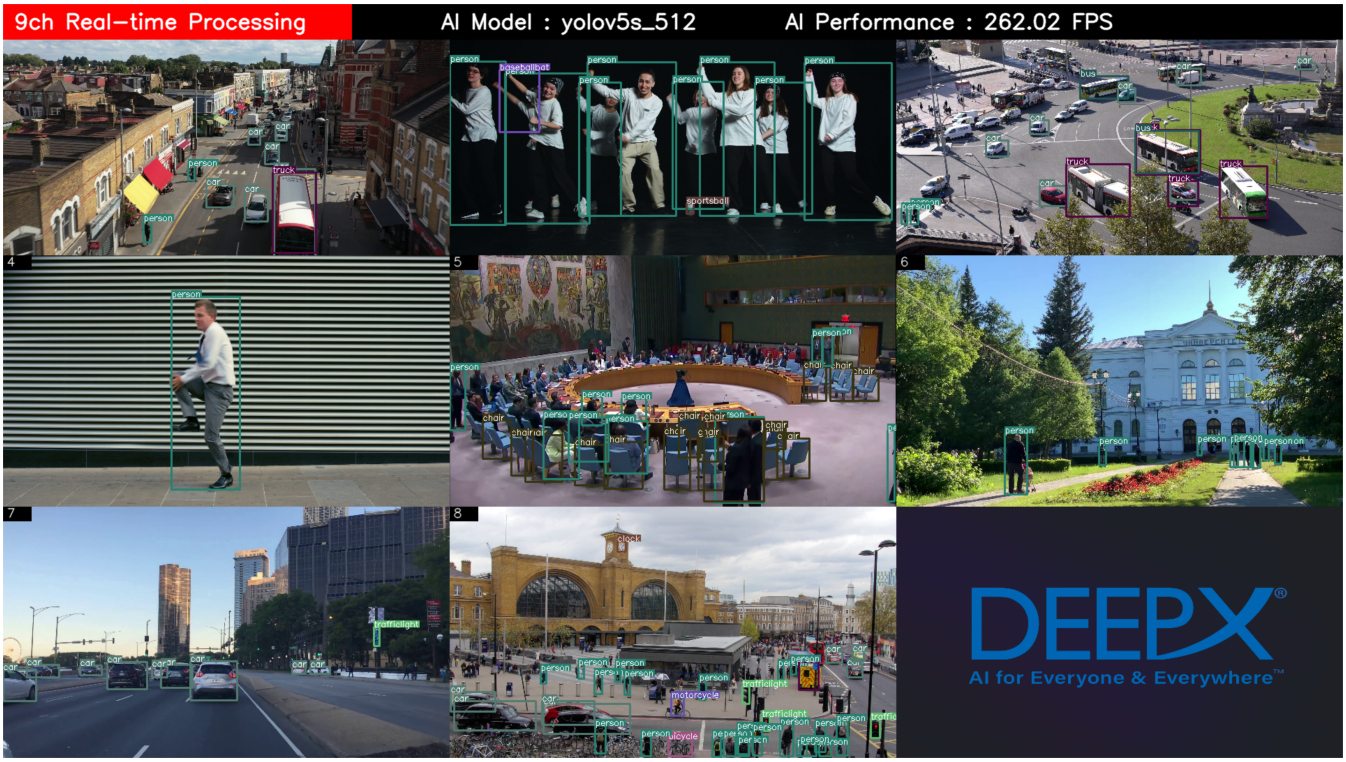
```
        createYoloLayerParam("439", 20, 20, 3, { 30.0f, 62.0f, 59.0f }, { 61.0f, 45.0f,
119.0f }, { 1 }),
        createYoloLayerParam("500", 10, 10, 3, { 116.0f, 156.0f, 373.0f }, { 90.0f,
198.0f, 326.0f }, { 2 })
    },
        .
        .
        .
    },
    .classNames = {"person" ,"bicycle" ,"car" ,"motorcycle", . . .}
}
```

YOLO Object Detection - Multi Channel

This demo performs object detection on multiple input streams simultaneously using a YOLOv5 model.

- Output Example: Upon execution, detection results across all input streams will be displayed in a tiled or multi-channel format.

```
./bin/yolo_multi -c ./example/yolo_multi/yolo_multi_demo.json
```



JSON Configuration for Multi-Channel YOLO Demo

The multi-channel demo uses a JSON configuration file to to define the input sources and model behavior.

- File Location: `example/yolo_multi/yolo_multi_input_source_demo.json`

Edit this file to customize the input video sources and number of streams.

Offline Mode in Input Method

To run detection in offline mode (for video file)

- Set the `video_sources` field in the JSON file
- Specify the number of frames to process as the third parameter

```
{
    "usage": "multi",
        "model_path": "/model_path/yolov5s_512", "model_name": "yolov5s_512",

        "video_sources": [
            ["./assets/videos/dron-citry-road.mov", "realtime"],
            ["/dev/video0", "camera"],
            ["./sample/1.jpg", "image"],
            ["rtsp://210.99.70.120:1935/live/cctv010.stream", "rtsp"],
            ["./assets/videos/dance-group.mov", "offline", 400]
        ],

        "display_config": {
            "display_label": "output",
            "capture_period": 33,
            "output_width": 1920,
            "output_height": 1080,
            "show_fps": true
    }
}
```

Pre-processing and Post-processing Parameters

YOLO models in **DX-APP** require external configuration for pre-processing and post-processing parameters. These parameters are **not** embedded in the compiled `.dxnn` model file.

Pre-processing and post-processing parameters

- Defined in: `demos/demo_utils/yolo_cfg.cpp`
- Referenced in: `yolo_multi_channels.cpp`

To customize,

- Modify the existing parameters in both files, or
- Add new parameter entries with a new model name to both files.

```
YoloParam yolov5s_512 = {
    512,  // height
    512,  // width
    0.25, // confThreshold
    0.3,  // scoreThreshold
    0.4,  // iouThreshold
    0,   // numBoxes
```

```
    80,   // numClasses
    "output", // onnx output name
    {     // if use_ort = off, layers config
        createYoloLayerParam("378", 40, 40, 3, { 10.0f, 16.0f, 33.0f }, { 13.0f, 30.0f,
23.0f }, { 0 }),
        createYoloLayerParam("439", 20, 20, 3, { 30.0f, 62.0f, 59.0f }, { 61.0f, 45.0f,
119.0f }, { 1 }),
        createYoloLayerParam("500", 10, 10, 3, { 116.0f, 156.0f, 373.0f }, { 90.0f,
198.0f, 326.0f }, { 2 })
    },
        .
        .
        .
    },
    .classNames = {"person" ,"bicycle" ,"car" ,"motorcycle", . . .}
}
```

```
YoloParam getYoloParameter(string model_name){
    if(model_name == "yolov5s_320")
        return yolov5s_320;
    else if(model_name == "yolov5s_512")
        return yolov5s_512;
    else if(model_name == "yolov5s_640")
        return yolov5s_640;
    else if(model_name == "yolox_s_512")
        return yolox_s_512;
    else if(model_name == "yolov7_640")
        return yolov7_640;
    else if(model_name == "yolov7_512")
        return yolov7_512;
    else if(model_name == "yolov4_608")
        return yolov4_608;
    return yolov5s_512;
}
```

RTSP Stream Input

To run a demo using an **RTSP** video stream, configure the input source in the JSON file as follows.

- Set the **RTSP** stream address under the `video_sources` field
- Set the network type to `rstp`

```
{
    .
    .
    .
    "video_sources": [
        ["rtsp://your_rtsp_stream_address", "rtsp"]
    ],
    .
    .
```

```
    .
}
```

This enables real-time inference directly from network video streams using DEEPX NPU.

3.2.3 Pose Estimation

This section explains how to run pose estimation demos based on Ultralytics YOLO Pose model using DEEPX NPU.

- Model: Ultralytics YOLO Pose (`YOLOV5Pose640_1.dxnn`)

Pose Estimation

```
./bin/pose -m assets/models/YOLOV5Pose640_1.dxnn -i sample/7.jpg -p 0
```

The YOLO pose model predicts human key points for each detected person within an image or video frame.

Pre-processing and Post-processing Parameters
Yolo Pose models do not embed external pre-processing and post-processing parameters in the compiled `.dxnn` file.

- Parameters defined in: `demos/demo_utils/yolo_cfg.cpp`
- Parameters referenced in: `yolo_pose.cpp`

```
// pre/post parameter table
YoloParam yoloParams[] = {
    [0] = yolov5s6_pose_640, // ----> p option : 0
};
```

To configure custom pose estimation parameters

- Modify the existing entries in the configuration files, or
- Add new model-specific configurations

This setup ensures that the NPU output is properly decoded into keypoint coordinates and bounding boxes.

3.2.4 Segmentation

This section explains how to run semantic segmentation demos based on YOLOv5 models. Both semantic segmentation models are supported.

- Model: `DeepLabV3PlusMobileNetV2_2.dxnn`

Semantic Segmentation - Cityscape Dataset-Based Demo
This demo describes an example of semantic segmentation based on the DeepLabV3Plus model trained on the **Cityscape** dataset, designed for detailed urban scene parsing.

- Model: `DeepLabV3PlusMobileNetV2_2.dxnn`
- Path: `assets/models`
- Dataset: Trained on the **Cityscape** dataset

This model performs pixel-wise classification, assigning a semantic label to each pixel in the input image. This allows for dense and structured scene understanding.

Notes.

- Load `DeepLabV3PlusMobileNetV2_2.dxnn` in the segmentation pipeline.
- Use the **Cityscape** class index mappings to interpret the output mask.

```
/* class_index, class_name, colorB, G, R */
SegmentationParam segCfg[] = {
    { 0 , "road" , 128 , 64 , 128 , },
    { 1 , "sidewalk" , 244 , 35 , 232 , },
    { 2 , "building" , 70 , 70 , 70 , },
    { 3 , "wall" , 102 , 102 , 156 , },
    { 4 , "fence" , 190 , 153 , 153 , },
    { 5 , "pole" , 153 , 153 , 153 , },
    { 6 , "traffic light" , 51 , 255 , 255 , },
    { 7 , "traffic sign" , 220 , 220 , 0 , },
    { 8 , "vegetation" , 107 , 142 , 35 , },
    { 9 , "terrain" , 152 , 251 , 152 , },
    { 10 , "sky" , 255 , 0 , 0 , },
    { 11 , "person" , 0 , 51 , 255 , },
    { 12 , "rider" , 255 , 0 , 0 , },
    { 13 , "car" , 255 , 51 , 0 , },
    { 14 , "truck" , 255 , 51 , 0 , },
    { 15 , "bus" , 255 , 51 , 0 , },
    { 16 , "train" , 0 , 80 , 100 , },
    { 17 , "motorcycle" , 0 , 0 , 230 , },
    { 18 , "bicycle" , 119 , 11 , 32 , },
};
```

```
./bin/segmentation -m assets/models/DeepLabV3PlusMobileNetV2_2.dxnn -i sample/8.jpg
```

Semantic Segmentation - Object Detection Demo

This demo explains segmentation capabilities by combining semantic segmentation and object detection.

- Segmentation Model: `DeepLabV3PlusMobileNetV2_2.dxnn`
- Object Detection Model: YOLOv5

How It Works,

- The segmentation model classifies every pixel into semantic categories such as road, person, or building
- The detection model locates and classifies objects such as e.g., cars, traffic signs

Benefits of Combined Pipeline

- Semantic segmentation provides context-aware, fine-grained understanding of background and scene layout
- Object detection focuses on identifying and locating distinct object instances

```
./bin/od_segmentation -m0 assets/models/YOLOV5S_3.dxnn -p0 1 -m1 assets/models/
DeepLabV3PlusMobileNetV2_2.dxnn -i sample/8.jpg
```

4. Template Guide

This section describes how to run classification and object detection tasks using the **DX-APP** application template system. Templates allow you to execute demos by simply modifying a JSON configuration file —no code changes are required.

Currently, the supported application templates are Classification and YOLO-based Object Detection.

Note. If your YOLO model requires custom decoding logic, you **must** manually update the yoloCustomDecode function located in `lib/utils/ box_decode.hpp`.

Related Sections are as follows.

- Section. Classification Templates
- Section. Object Detection Templates
- Section. Python Examples Templates

4.1 Classification Template

Here is an example of a classification template.

```
./bin/run_classifier -c example/run_classifier/imagenet_example.json
...
[example/ILSVRC2012/0.jpeg] Top1 Result : class 831 (studio couch, day bed)
[example/ILSVRC2012/1.jpeg] Top1 Result : class 321 (admiral)
[example/ILSVRC2012/2.jpeg] Top1 Result : class 846 (table lamp)
[example/ILSVRC2012/3.jpeg] Top1 Result : class 794 (shower curtain)
```

The `example/run_classifier/imagenet_example.json` file is a sample configuration for running a classification model. You can use this file as a reference to customize input and output settings, and optionally modify the application section to control how classification results are displayed or saved as follows.

```
"application": {
    "description": "result of post-processing after inference, you can use \"save\" or
\"realtime\" or \"none\"", "type": "save"
}
```

For a detailed explanation of how classification works, refer to **Section. Classification Template Guide**.

4.2 Object Detection Template

Here is an example of an objection detection template.

```
./bin/run_detector -c example/run_detector/yolov5s3_example.json
```

Since the application is running in 'save' mode, five result images will be saved in the current directory with names like `result-app1.jpg`, `result-app2.jpg`, `result-app3.jpg`, `result-app4.jpg`, and `result-app5.jpg`.

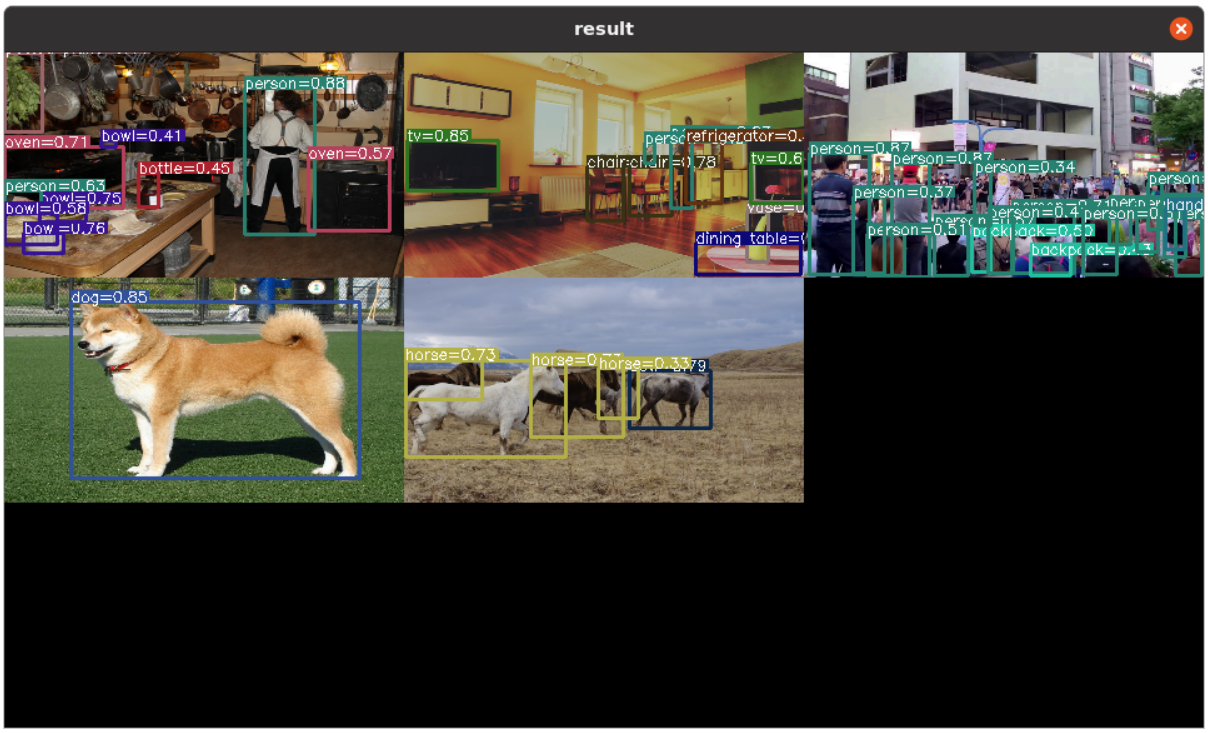
The `example/run_detector/yolov5s3_example.json` file is a sample configuration for running an object detection model. You can use this file as a reference to customize the input and output settings, and optionally modify the application section to control how detection results are displayed and saved.

```
"application": {
    "description": "result of post-processing after inference, you can use \"save\" or
\"realtime\" or \"none\"",
    "type": "realtime"
}
```

The `type` field in the JSON configuration defines how detection results are handled.

- `save` : Saves the output file as a video file
- `realtime` : Displays the output on the screen
- `none` : Prints the count of detection results continuously to the terminal

In the previous example, the json file sets `type` as `none` .
If you want to save the output, change it to `save` .



When using a YOLO model, post-processing parameters can be customized based on the official YOLOv5 configuration. You may also modify the class information directly in the JSON file, so no code recompilation is required.

The `yolo_basic` method supports standard decoding for YOLOv3, YOLOv5, and YOLOv7. Other supported decoding types include `yolo_scale` , `yolox` , `yolo_pose` , and `yolo_face` .

Refer to the corresponding JSON configuration file for detailed information.
You can check the output format information by running "run_model -m {model_path}.dxnn {--use-ort}".

```
"model":{
    "path":"assets/models/YOLOV5S_3.dxnn",
    "param":{
            "name": "YOLOV5S_3",
            "input_width" : 512,
            "input_height" : 512,
            "score_threshold": 0.3,
            "iou_threshold": 0.4,
```

```
            "last_activation": "sigmoid",
            "decoding_method": "yolo_basic",
            "box_format": "center",
            "final_outputs":[
                "output"
            ],
            "layer": [
                {
                    "name": "output",
                    "shape":[1, 16128, 85]
                },
                {
                    "name": "378",
                    "stride": 8,
                    "anchor_width": [10, 16, 33],
                    "anchor_height": [13, 30, 23]
                },
                {
                    "name": "439",
                    "stride": 16,
                    "anchor_width": [30, 62, 59],
                    "anchor_height": [61, 45, 119]
                },
                {
                    "name": "500",
                    "stride": 32,
                    "anchor_width": [116, 156, 373],
                    "anchor_height": [90, 198, 326]
                }
        ]
    }
}
```

The template also supports multi-channel detection, allowing multiput input streams (i.e., images files, video files, and camera devices) to be processed simultaneously.

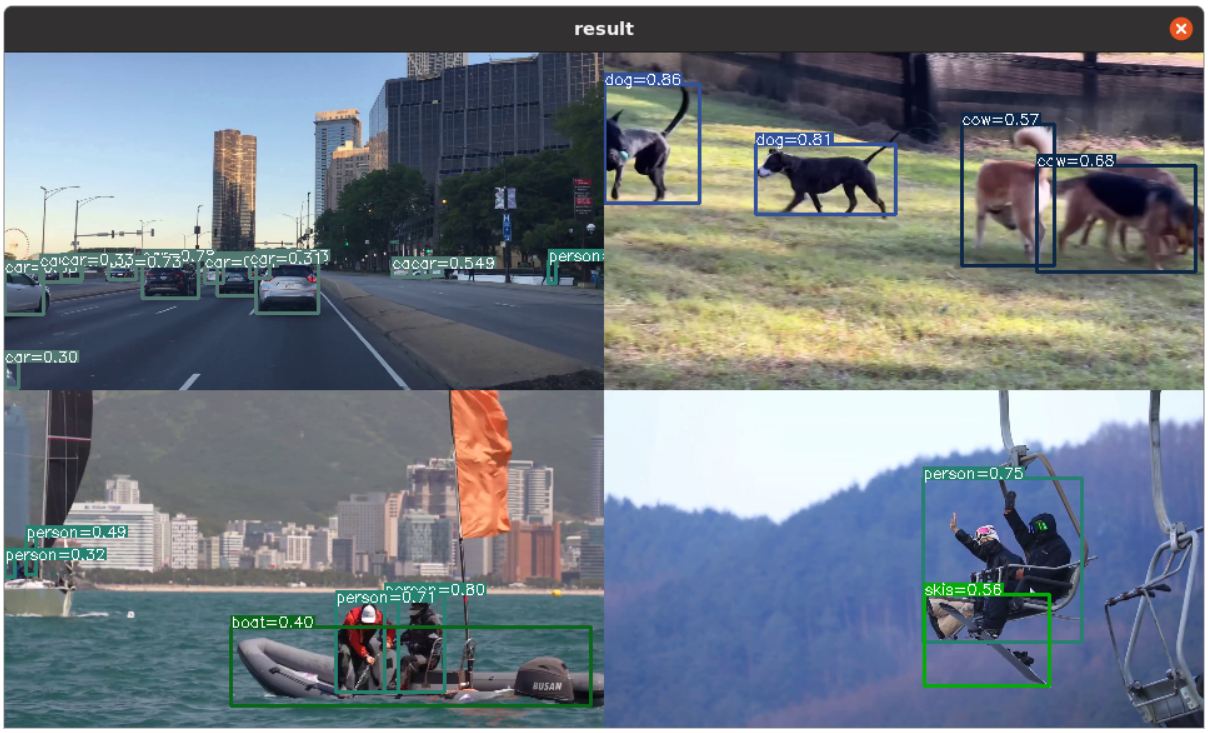
To enable this, list the input sources in the `sources` field of the `input` section in your JSON configuration. The output from each channel will be displayed in a checkerboard layout.

When `type` in the `sources` field of the `input` section is set to `video`, you can additionally specify the number of frames to process. This provides control over how many frames are used for pre-processing and inference.

```
"input":{
    "format":"RGB",
    "sources":[
        {
            "type":"image", "path":"/your-sample-image-path/1.jpg"
        },
        {
            "type":"video", "path":"/your-sample-video-path/1.mp4"
        },
```

```
        {
            "type":"video", "path":"/your-sample-video-path/1.mp4", "frames":100
        },
        {
            "type":"camera", "path":"/dev/video0"
        }
    ]
},
```

```
./bin/run_detector -c example/run_detector/yolov5s3_realtime_example.json
```



If you are using a custom YOLO model that does **not** follow the standard decoding logic, or if you are **not** using the `yolo_basic` decoding method, you **must** implement a custom decode function.

You can use the existing yoloXDecode function in `lib/utils/box_decode.hpp` as a reference. Analyze the decoding logic and write your own implementation tailored to your model's output tensor format.

- Path: `lib/utils/box_decode.hpp`

```
dxapp::common::BBox yoloCustomDecode(std::function<float(float)> activation,
std::vector<float> datas, dxapp::common::Point grid, dxapp::common::Size
anchor, int stride, float scale)
{
    /**
        * @brief adding your decode method
        *
        * example code ..
        * dxapp::common::BBox box_temp;
        * box_temp._xmin = (activation(datas[0]) * 2. - 0.5 + grid._x ) * stride; //
```

```
center x
        * box_temp._ymin = (activation(datas[1]) * 2. - 0.5 + grid._y ) * stride; //
center y
        * box_temp._width = std::pow((activation(datas[2]) * 2.f), 2) * anchor._width;
        * box_temp._height = std::pow((activation(datas[3]) * 2.f), 2) * anchor._height;
        * dxapp::common::BBox result = {
        *     ._xmin=box_temp._xmin - box_temp._width / 2.f,
        *     ._ymin=box_temp._ymin - box_temp._height / 2.f,
        *     ._xmax=box_temp._xmin + box_temp._width / 2.f,
        *     ._ymax=box_temp._ymin + box_temp._height / 2.f,
        *     ._width = box_temp._width,
        *     ._height = box_temp._height,
        * };
        *
    */

    dxapp::common::BBox result;

    return result;
};
```

Note. Once you modify the code, you **must** recompile it.

```
./build.sh
```

For a detailed explanation of how to write your models' custom post-processing, refer to **Section. Object Detection Template Guide**.

5. Classification Template

This chapter describes how to use the classification template to run image classification or regression models on DEEPX NPU. This workflow does **not** require source code modification and only a JSON configuration file needs to be modified.

It is important to fully understand the output structure of your model, as output shapes may vary depending on model type and NPU compilation settings.

Output Post-Processing
For faster result interpretation, it is recommended to include an ArgMax layer at the end of the model to directly output class indices.

5.1 Run Classification Template

This section explains how to execute an image classification task using a DXNN-compiled model and a JSON configuration file.

Configuration File
The template uses a JSON configuration file to define input/output settings and runtime parameters.
A sample configuration is provided `example/run_classifier/imagenet_example.json`.

How to Run
To execute the demo, run as follows.

```
./bin/run_classifier -c example/run_classifier/imagenet_example.json
...
[sample/ILSVRC2012/0.jpeg] Top1 Result : class 905 (window shade)
[sample/ILSVRC2012/1.jpeg] Top1 Result : class 321 (admiral)
[sample/ILSVRC2012/2.jpeg] Top1 Result : class 846 (table lamp)
[sample/ILSVRC2012/3.jpeg] Top1 Result : class 794 (shower curtain)
...
```

Customization
You can modify the JSON file to

- Change input dimensions, normalization, or format
- Adjust output processing parameters
- Enable result display or configure file-based result saving

No source code changes are required. The entire configuration is driven by the JSON file.

5.2 Classification Post-Processing

5.2.1 Using ArgMax Layer

For classification models where only the Top-1 prediction is required, it is recommended to include an ArgMax layer at the end of the model. The provided example model, `EfficientNetB0_4`, is configured with an ArgMax layer.

Output Format
When using an ArgMax-based model,

- The NPU output consists of a single `uint16_t` (2 bytes) value.
- This value directly represents the class index with the highest probability.

```
Task[0] npu_0, NPU, 8209728bytes (input 157696, output 2)
  inputs
    data, INT8, [1, 224, 224, 3, ], 0
  outputs
    argmax_output, UINT16, [1, ], 0
```

This value can be used as follows.

```
auto outputs = ie.Run(input_tensor);
auto max_index = *(uint16_t*)outputs.front()->data();
std::cout << "argmax : " << max_index << std::endl;
```

5.2.2 Using GAP or FC Layer

If the model's last layer is **not** ArgMax but a GAP (Global Average Pooling) or FC (Fully Connected) layer, the output is expected to be in the format `[1, 1, 1, number of classes]`.

- The following logic applies Softmax on the valid portion of the output.
- Reference: `lib/post_process/classification_post_processing.hpp` - Line 65

```
#include "post_process/classification_post_processing.hpp"
  .
  .
  .
  // n : number of classes
```

```
  auto outputs = ie.Run(input_tensor);
  std::vector<float> scores = classification::getSoftmax((float*)outputs.back()->data(),
n);
```

6. Object Detection Template

This section explains how to use the Object Detection Template to execute YOLO-series models on DeepX NPUs by configuring only a JSON file, thus no source code modifications are needed.

6.1 Run Object Detection Template (Yolo Model)

To run an object detection model using YOLOv5.

```
./bin/run_detector -c example/run_detector/yolov5s3_example.json
  ...
  detected : 9
```

Configuration File

- Path: `example/run_detector/yolov5s3_example.json`
- This file defines all runtime parameters including model path, input/output shape, post-processing method, and output type

Output Display Options

- `type value` : Description
- `save` : Saves output to a video file
- `realtime` : Displays results in a window (GUI)
- `none` : Continuously prints detection count (default)

Note. The current example file sets `type: none`. To save video output, change it to `save`.

Supported Post-Processing Methods
The detection template supports various decoding methods tailored to different YOLO variants.

- `yolo_basic` : For YOLOv3, YOLOv5, YOLOv7
- `yolo_scale` : Multi-scale YOLO
- `yolox` : For YOLOX
- `yolo_pose` : For human pose estimation
- `yolov8` : For YOLOv8 models
- `yolo_face` : For face detection models

Post-processing parameters, such as thresholds and class names, can be adjusted in the JSON file without recompilation.

6.2 Custom Post-Processing Guide for Your Models

This guide provides instructions for customizing the post-processing pipeline to suit your model architecture and deployment configuration.
Post-processing behavior may vary depending on the model's output shape and the DeepX NPU execution environment. It is essential to understand your model's structure and configure post-processing accordingly.

Supported scenarios include

- Post-Processing with `USE_ORT=ON`
- Post-Processing with `USE_ORT=OFF`
- (Optional) Custom Post-Processing

6.3 Post-Processing Using USE_ORT=ON

If the DXRT framework is executed with `USE_ORT=ON`, the output will be in the same format as the original ONNX model. Therefore, you can directly apply your existing ONNX post-processing pipeline.

And Applying NMS
After struct conversion, the `cxcywh` format **must** be translated to corner format (`xmin, ymin, xmax, ymax`). This enables accurate rendering and overlap calculations.

Following decoding, apply Non-Maximum Suppression (NMS) to remove overlapping bounding boxes and retain only high-confidence predictions.

Refer to `dx_app/lib/post_process/yolo_post_processing.hpp` - line.204 for the Reference Implementation.

6.4 Post-Processing Using USE_ORT=OFF and No PPU

If PPU is **disabled** and the model is executed with `USE_ORT=OFF`, the model output will consist of three blobs.

```
inputs
    images, UINT8, [1, 512, 512, 3, ], 0

outputs
    378, FLOAT, [1, 255, 64, 64, ], 0
    439, FLOAT, [1, 255, 32, 32, ], 0
    500, FLOAT, [1, 255, 16, 16, ], 0
```

Required Post-Processing Steps

- **1.** Anchor Box Scaling and Offset Recovery: Reconstruct the predicted bounding box coordinates using the anchor box configuration and grid position.
- **2.** Sigmoid Activation: Apply the sigmoid function to bounding box offsets , objectness scores, and class probabilities.
- **3.** Non-Maximum Suppression (NMS): Filter overlapping boxes and retain only the highest-confidence detections.

Refer to `dx_app/lib/post_process/yolo_post_processing.hpp` for implementation details.

6.5 (Optional) Custom Post-Processing

If you are working with a custom detection model that does not follow the standard YOLO output structure, you can define your own post-processing logic.

Step 1. Set custom_decode in JSON
In your configuration file (e.g., `yolov5s3_example.json`), set the decoding_method field to `decoding_method : custom_decode`. This tells the SDK to bypass default decoding and call your custom implementation.

Step 2. Modify getBoxesFromCustomPostProcessing()

Update the following function to handle your model's output format

- File: `dx_app/lib/post_process/yolo_post_processing.hpp` - line.630

You can add any necessary parameters here.

```
void getBoxesFromCustomPostProcessing(uint8_t* outputs /* Users can add necessary
parameters manually. */)
{
    /**
        * @brief adding your post processing code
        *
        * example code ..
        *
        * int boxIdx = 0;
        * float* node_a = (float*)outputs;
        * float* node_b = (float*)node_a + node_a_size;
        * float* node_c = (float*)node_b + node_b_size;
        * for(int i=0; i<node_length; i++)
        * {
        *
        * }
        *
        *
        */
};
```

Step 3. (Optional) Modify yoloCustomDecode()

For more advanced control, modify the low-level decoding logic in `dx_app/lib/utils/box_decode.hpp`. You can use this function to implement custom decoding logic beyond the default YOLO box structure.

```
dxapp::common::BBox yoloCustomDecode(std::function<float(float)> activation,
std::vector<float> datas, dxapp::common::Point grid, dxapp::common::Size
anchor, int stride, float scale)
{
    /**
      * @brief adding your decode method
      *
      * example code ..
          * dxapp::common::BBox box_temp;
          * box_temp._xmin = (activation(datas[0]) * 2. - 0.5 + grid._x ) * stride; //
center x
          * box_temp._ymin = (activation(datas[1]) * 2. - 0.5 + grid._y ) * stride; //
center y
          * box_temp._width = std::pow((activation(datas[2]) * 2.f), 2) * anchor._width;
          * box_temp._height = std::pow((activation(datas[3]) * 2.f), 2) *
anchor._height;
          * dxapp::common::BBox result = {
              * ._xmin=box_temp._xmin - box_temp._width / 2.f,
              * ._ymin=box_temp._ymin - box_temp._height / 2.f,
```

```
            * ._xmax=box_temp._xmin + box_temp._width / 2.f,
            * ._ymax=box_temp._ymin + box_temp._height / 2.f,
            * ._width = box_temp._width,
            * ._height = box_temp._height,
      * };
      *
      */

    dxapp::common::BBox result;

    return result;

};
```

Note. Once you modify the code, you **must** recompile it.

```
./build.sh
```

7. Python Examples

This section describes how to run inference and post-processing on DeepX NPUs using Python. It includes examples for image classification and object detection (YOLO series).

7.1 Python Example Guide

This section provides a step-by-step guide to implementing Python-based preprocessing, inference, and post-processing for DeepX NPUs. It covers the full workflow—from installing the dx_engine package and preparing aligned input data to running inference and handling model outputs.

Step 1. Install dx_engine
To Install the `dx_engine` Python Package, navigate to the `python_package` folder inside the `dx_rt` directory and run the following command.

```
cd /your-dxrt-directory/python_package
pip install .
```

Example output

```
Successfully built dx-engine
Installing collected packages: dx-engine
  Attempting uninstall: dx-engine
    Found existing installation: dx-engine 1.0.0
    Uninstalling dx-engine-1.0.0:
      Successfully uninstalled dx-engine-1.0.0
Successfully installed dx-engine-1.0.0
```

Step 2. Import dx_engine Python Module
Start by importing the necessary module.

```
from dx_engine import InferenceEngine
```

Step 3. Install Python Requirements
Before running any Python templates, install the required dependencies:

```
pip install -r ./templates/python/requirements.txt
```

Step 4. Create an Input Tensor

First, create an inference engine instance by specifying the model path. Then, generate an input tensor based on the model's input size.

```
import numpy as np
from dx_engine import InferenceEngine

ie = InferenceEngine("assets/models/YOLOV5S_3.dxnn")
input_tensor = np.array(ie.input_size(), dtype=np.uint8)
# Adjust the type according to the input data you are using
```

Step 5. Preparing an Image for Inference

Prepare and adjust an input image to meet the requirements of the NPU.

```
import numpy as np
import cv2
from dx_engine import InferenceEngine

def preprocessing(image, new_shape=(224, 224), format=None):
    image = cv2.resize(image, new_shape)
    h, w, c = image.shape
    if format is not None:
        image = cv2.cvtColor(image, format)
    return image

ie = InferenceEngine("assets/models/EfficientNetB0_4.dxnn")
image_src = cv2.imread("images/1.jpg", cv2.IMREAD_COLOR)

image_input = preprocessing(image_src, new_shape=(224, 224), align=align, format =
cv2.COLOR_BGR2RGB)
```

7.1.1 Run ImageNet Python Example (Classification)

This section demonstrates how to run image classification examples using Python on DeepX NPUs.

To check the usage and available options: `python imageNet_example.py --help`

Classification

```
python template/python/imageNet_example.py --config example/run_classifier/
imagenet_example.json
  ...
  [example/ILSVRC2012/0.jpeg] Top1 Result : class 831 (studio couch, day bed)
  [example/ILSVRC2012/1.jpeg] Top1 Result : class 321 (admiral)
  [example/ILSVRC2012/2.jpeg] Top1 Result : class 846 (table lamp)
  [example/ILSVRC2012/3.jpeg] Top1 Result : class 794 (shower curtain)
```

Steps

- **1.** Import The InferenceEngine module which is a callable inference engine module.

```
from dx_engine import InferenceEngine
```

- **2.** Initialize the Inference Engine with `InferecneEngine` module.

```
ie = InferenceEngine("./assets/models/EfficientNetB0_4.dxnn")
```

- **3.** Prepare the Input Tensor

    Input and output tensor shapes follow the format: [N, H, W, C]

Output Handling

Using ArgMax Layer

- The provided `EfficientNetB0_4` model ends with an ArgMax layer
- Output: a single `uint16_t` value (2 bytes) representing the Top-1 class index

```
Task[0] npu_0, NPU, 8209728bytes (input 157696, output 2)
  inputs
    data, INT8, [1, 224, 224, 3, ], 0
  outputs
    argmax_output, UINT16, [1, ], 0
```

This value can be used as follows.

```
ie_output = ie.Run(image_input)
output = ie_output[0][0]
print("[{}] Top1 Result : class {} ({})".format(input_path, output, classes[output]))
```

Using GAP or FC Layer

If the final layer is a Global Average Pooling (GAP) or Fully Connected (FC) layer.

- Output shape: `[1, 1, 1, num_classes]`

Apply Softmax to convert logits into class probabilities

```
def postprocessing(outputs, n_classes):
    outputs = outputs[...,:n_classes]
    exp_result = np.exp(outputs - np.max(outputs))
    exp_result = exp_result / exp_result.sum()
    top1 = np.argmax(exp_result)
    return top1
```

```
    .
    .
    .
    ie_output = ie.Run(image_input)
    output = postprocessing(ie_output[0], len(classes))
    print("[{}] Top1 Result : class {} ({})".format(input_path, output,
classes[output]))
```

7.1.2 Run YoloV5S Python Example (Object Detection)

This section demonstrates how to run object detection examples using Python on DeepX NPUs.

To see available options for detection: `python yolov5s_example.py --help`

Object Detection

```
python template/python/yolov5s_example.py
```

or

```
python template/python/yolov5s_example.py --config example/run_detector/
yolov5s3_example.json
  ...
  [Result] Detected 10 Boxes.
  [0] conf, classID, x1, y1, x2, y2, : 0.8771, person(0), 307, 139, 401, 364
  [1] conf, classID, x1, y1, x2, y2, : 0.7358, bowl(45), 46, 317, 107, 347
  [2] conf, classID, x1, y1, x2, y2, : 0.7192, bowl(45), 25, 360, 79, 393
  [3] conf, classID, x1, y1, x2, y2, : 0.6766, oven(69), 0, 218, 154, 325
  [4] conf, classID, x1, y1, x2, y2, : 0.5811, oven(69), 389, 246, 497, 359
  [5] conf, classID, x1, y1, x2, y2, : 0.5664, person(0), 0, 295, 48, 332
  [6] conf, classID, x1, y1, x2, y2, : 0.5365, bowl(45), 1, 329, 69, 380
  [7] conf, classID, x1, y1, x2, y2, : 0.4199, potted plant(58), 0, 86, 50, 206
  [8] conf, classID, x1, y1, x2, y2, : 0.3649, bottle(39), 172, 271, 203, 323
  [9] conf, classID, x1, y1, x2, y2, : 0.3084, cup(41), 117, 300, 137, 327
  ...
```

Steps

To do this, modify the Python application as needed, referring to the example implementation in `./python/yolov5s_example.py`.

- **1.** Import the Inference Module which is a callable inference engine module.

```
from dx_engine import InferenceEngine
```

- **2.** Initialize with YOLO Model with `InferecneEngine` module.

```
ie = InferenceEngine("./assets/models/YOLOV5S_3.dxnn")
```

- **3.** Understand Channel Alignment

YOLO models typically have 255 output channels

- (80 classes + 1 objectness + 4 bbox) × 3 anchors = 255
- **4.** Decode the Output

Use the `all_decode()` function in `python/yolov5s_example.py` - lines 74–102
This handles decoding bounding boxes, class scores, and filtering

```
image_src = cv2.imread(input_path, cv2.IMREAD_COLOR)
image_input, _, _ = letter_box(image_src, new_shape=(int(input_size), int(input_size)),
fill_color=(114, 114, 114), format=cv2.COLOR_BGR2RGB)

''' detect image (1) run dxrt inference engine, (2) post processing'''
ie_output = ie.Run(image_input)
```

```
print("dxrt inference Done! ")
decoded_tensor = []
if ie.output_dtype()[0] == "BBOX":
    decoded_tensor = ppu_decode(ie_output, layers)
elif len(ie_output) > 1:
    cpu_model_path = os.path.join(os.path.split(model_path)[0], "cpu_0.onnx")
    if os.path.exists(cpu_model_path):
        decoded_tensor = onnx_decode(ie_output, cpu_model_path)
    else:
        decoded_tensor = all_decode(ie_output, layers)
else:
    decoded_tensor = ie_output[0]
print("decoding output Done! ")

''' post Processing '''
x = np.squeeze(decoded_tensor)
x = x[x[..., 4]>score_threshold]
box = ops.xywh2xyxy(x[..., :4])
x[:,5:] *= x[:,4:5]
conf = np.max(x[..., 5:], axis=-1, keepdims=True)
j = np.argmax(x[..., 5:], axis=-1, keepdims=True)
mask = conf.flatten() > score_threshold
filtered = np.concatenate((box, conf, j.astype(np.float32)), axis=1)[mask]
sorted_indices = np.argsort(-filtered[:, 4])
x = filtered[sorted_indices]
x = torch.Tensor(x)
x = x[torchvision.ops.nms(x[:,:4], x[:, 4], score_threshold)]

''' save result and print detected info '''
print("[Result] Detected {} Boxes.".format(len(x)))
image = cv2.cvtColor(image_input, cv2.COLOR_RGB2BGR)
colors = np.random.randint(0, 256, [80, 3], np.uint8).tolist()
for idx, r in enumerate(x.numpy()):

    pt1, pt2, conf, label = r[0:2].astype(int), r[2:4].astype(int), r[4],
r[5].astype(int)
    print("[{}] conf, classID, x1, y1, x2, y2, : {:.4f}, {}({}), {}, {}, {}, {}"
        .format(idx, conf, classes[label], label, pt1[0], pt1[1], pt2[0], pt2[1]))
    image = cv2.rectangle(image, pt1, pt2, colors[label], 2)
cv2.imwrite("yolov5s.jpg", image)
print("save file : yolov5s.jpg ")
```

8. YOLO Post-Processing (Pybind11)

This chapter focuses on accelerating YOLO post-processing in Python using Pybind11. It introduces the `dx_postprocess` module and the `YoloPostProcess` class, lists supported models, and explains synchronous/asynchronous usage with examples.

8.1 YOLO Post-Processing Optimization Using Pybind11

8.1.1 Overview

Post-processing for YOLO models can be implemented in Python using libraries such as numpy and `torchvision.ops.nms`. This is demonstrated in `templates/python/yolov5s_example.py` and `templates/python/yolov_async.py`.

However, these Python-based implementations may not offer optimal performance in terms of latency, particularly for real-time applications.

To maximize FPS (Frames Per Second) in time-sensitive scenarios, it's recommended to implement performance-critical post-processing steps—such as score filtering, bounding box decoding, and Non-Maximum Suppression (NMS)—in C++, and integrate them into Python using Pybind11.

This approach significantly reduces latency by offloading heavy computations to native C++ code while maintaining the usability of Python.

The typical YOLO post-processing pipeline consists of the following steps.

- Score filtering
- Bounding box decoding
- NMS (Non-Maximum Suppression)

These steps involve intensive matrix operations and iterative loops, making them performance-critical. Executing them in C++ is significantly more efficient than in Python, especially for real-time applications.

To address this, DX-APP provides the **`dx_postprocess`** module, which includes the **`YoloPostProcess`** class, an optimized C++ implementation of YOLO post-processing exposed to Python with Pybind11.

By using **`YoloPostProcess`**, users can perform end-to-end YOLO post-processing with a single function call, supporting a variety of YOLO model types with minimal effort and maximum performance.

8.1.2 Supported Use Cases for YoloPostProcess

`YoloPostProcess` demonstrates how C++ code can be wrapped with Pybind11 to optimize post-processing in Python environments. It shows how to implement performance-critical post-processing steps (score filtering, bounding box decoding, NMS, etc.) in C++ and make them available for use in Python.

Important: `YoloPostProcess` is not a universal solution that supports post-processing for all YOLO series models. It is an example implementation designed to work only with specific models and configurations.

Among the models provided by DX-APP, the following cases support `YoloPostProcess`:

Model	USE_ORT = ON	USE_ORT = OFF	Config
YOLOV3_1	O	O	YOLOV3_1.json
YOLOV4_3	O	X	YOLOV4_3.json
YOLOV5Pose640_1	O	X	YOLOV5Pose640_1.json
YOLOV5S_1	O	O	YOLOV5S_1.json
YOLOV5S_3	O	O	YOLOV5S_3.json
YOLOV5S_4	O	O	YOLOV5S_4.json
YOLOV5S_6	O	O	YOLOV5S_6.json
YOLOV5S_Face-1	O	X	YOLOV5S_Face-1.json
YOLOV5X_2	O	O	YOLOV5X_2.json
YOLOv7_512	O	O	YOLOV7_512.json
YoloV7	O	O	YoloV7.json
YoloV8N	O	X	YoloV8N.json
YOLOV9S	O	X	YOLOV9S.json
YOLOX-S_1	O	X	YOLOX-S_1.json

Preconfigured JSON config files for supported YOLO models are available in the test/data directory.

Alternative for unsupported cases: For models or configurations not supported by `YoloPostProcess`, you can refer to the `lib/pybind` codes to implement custom C++ post-processing code and wrap it with

Pybind11. This allows you to create optimized post-processing tailored to the specific characteristics of your model.

8.1.3 Run YoloPostProcess Python Example

The following example demonstrates how to perform YOLO post-processing using the `YoloPostProcess` class from the `dx_postprocess` module.

```
$ python template/python/yolo_pybind_example.py --video_path /path/to/your/video_file --
config_path /path/to/your/config_file --run_async --visualize
```

Quick Example:

```
python templates/python/yolo_pybind_example.py --video_path assets/videos/dance-
group.mov --config test/data/YoloV7.json --run_async --visualize
```

This script takes a video file as input and performs Pre-processing, Inference, and Post-processing (synchronous or asynchronous).

After processing all frames, the average FPS (Frames Per Second) is calculated as:
FPS = Total Frames / Total Processing Time (in seconds)

Command-Line Arguments

- `--video_path`: Path to the input video file
- `--config_path`: Path to the JSON config file containing model path and post-processing parameters
- `--run_async`: Use asynchronous inference with `RunAsync()`, where post-processing is performed inside the callback function. If **not** specified, synchronous inference with `Run()` is used, and preprocessing, inference, and post-processing are executed sequentially for each frame.
- `--visualize`: Enables visualization of detection results

Post-Processing with `YoloPostProcess` Class

- **1.** Import the Module

```
from dx_postprocess import YoloPostProcess
```

- **2.** Initialize the Post-Processor
  Pass the config file path to initialize YoloPostProcess.

```
ypp = YoloPostProcess(json_config)
```

This sets up model-specific decoding parameters (e.g., class count, anchor sizes, thresholds).

For Synchronous Inference Mode
When `--run_async` is **not** specified,

- Pre-processing → Inference → Post-processing
- All steps run sequentially for each frame using `Run()`

```
# Initialize InferenceEngine
ie = InferenceEngine(json_config["model"]["path"])

# Pre-Processing
input_tensor, _, _ = letter_box(frame, (input_size, input_size), fill_color=(114, 114,
114), format=cv2.COLOR_BGR2RGB)

# Run the inference engine synchronously
ie_output = ie.Run(input_tensor)

# Post-Processing
pp_output = ypp.Run(ie_output)
```

For Asynchronous Inference Mode
When `--run_async` is specified,

- Uses `RunAsync()`
- Post-processing occurs inside the callback function, allowing pipelined execution and improved throughput

```
def pp_callback(ie_outputs, user_args):
    value:UserArgs = user_args.value
    pp_output_ = value.ypp_.Run(ie_outputs)
    q.put([value.input_tensor_, pp_output_])

class UserArgs:
```

```
    def __init__(self, ypp:YoloPostProcess, input_tensor):
        self.ypp_ = ypp
        self.input_tensor_ = input_tensor

# Initialize InferenceEngine
ie = InferenceEngine(json_config["model"]["path"])

# Register callback function
ie.RegisterCallBack(pp_callback)

# UserArgs for callback function
user_args = UserArgs(ypp, input_tensor)

# Pre-Processing
input_tensor, _, _ = letter_box(frame, (input_size, input_size), fill_color=(114, 114,
114), format=cv2.COLOR_BGR2RGB)

# Run inference asynchronously
req_id = ie.RunAsync(input_tensor, user_args)
```

9. Change Log

9.1 Version 2.0.0 (August 2025)

- Major code refactoring and restructuring of demo applications.
- Update on Tensor Index Assignment in Yolo Layer Reordering
- demos/demo_utils/yolo_cfg.cpp Structure Changes: To accommodate different post-processing methods based on the USE_ORT setting in RT, the final output name of the ONNX model is now received as a separate parameter.
- run_detector Updates: For more efficient post-processing, run_detector also now receives the final ONNX output name as a separate parameter.
- yolo_pybind_example.py Refactoring: Refactored the code to use a RunAsync() + Wait() structure instead of callbacks. This change ensures the correct handling of the output tensor order.
- Improved FPS calculation for the YOLO multi-demo.

9.2 Version 1.10.0 (June 2025)

- Initial create dx-app
- demo : classification
- demo : object detection
- demo : pose estimation
- demo : multi models for object detection and segmentation
- demo : semantic segmentation
- demo : multi channel oject detection
- template : classification
- template : object detection
- template : python example (sync/async/pybind c++)