如何用 OpenVINO™让 YOLOv8 获得 1000+ FPS 性能？

作者：武卓

YOLO 家族又添新成员了！作为目标检测领域著名的模型家族，you only look once (YOLO) 推 出新模型的速度可谓是越来越快。就在刚刚过去的 1 月份，YOLO 又推出了最新的 YOLOv8 模型，其模型结构和架构上的创新以及所提供的性能提升，使得它刚刚面世，就获得了广大开发者的关注。

YOLOv8 的性能到底怎么样？如果说利用 OpenVINO™的量化和加速，利用英特尔®CPU、集成显卡以及独立显卡与同一代码库无缝协作，可以获得 1000+ FPS 的性能，你相信吗？那不妨继续往下看，我们将手把手的教你在利用 OpenVINO™在英特尔®处理器上实现这一性能。

图 1. YOLOv8 推理结果示例

好的，让我们开始吧。

注意：以下步骤中的所有代码来自 OpenVINO Notebooks 开源仓库中的 230-yolov8-optimization notebook 代码示例，您可以点击以下链接直达源代码。
https://github.com/openvinotoolkit/openvino_notebooks/blob/main/notebooks/230-yolov8-optimization/230-yolov8-optimization.ipynb

第一步： 安装相应工具包及加载模型

本次代码示例我们使用的是 Ultralytics YOLOv8 模型，因此需要首先安装相应工具包。

```
1. !pip install "ultralytics==8.0.5"
```

然后下载及加载相应的 PyTorch 模型。

```
1. from ultralytics import YOLO
2.
3. MODEL_NAME = "yolov8n"
4.
5. model = YOLO(f'{MODEL_NAME}.pt')
6.
7. label_map = model.model.names
```

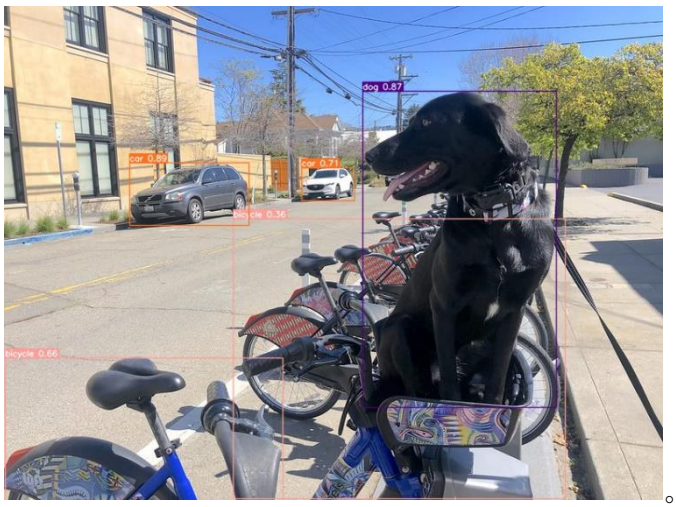
定义测试图片的地址，获得原始 PyTorch 模型的推理结果

```
1. IMAGE_PATH = "../data/image/coco_bike.jpg"
2. results = model(IMAGE_PATH, return_outputs=True)
```

其运行效果如下

```
Ultralytics YOLOv8.0.5 🚀 Python-3.8.10 torch-1.13.1+cpu CPU
Fusing layers...
YOLOv8n summary: 168 layers, 3151904 parameters, 0 gradients, 8.7 GFLOPs
```

为将目标检测的效果以可视化的形式呈现出来，需要定义相应的函数，最终运行效果如下图所示



。

第二步： 将模型转换为 OpenVINO IR 格式

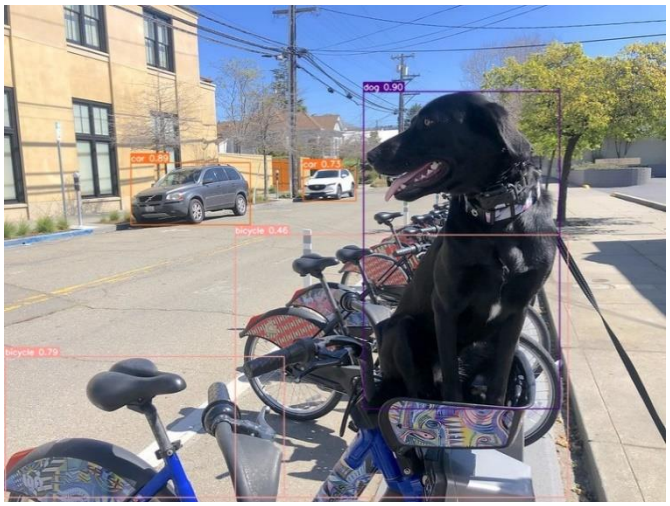
为获得良好的模型推理加速，并更方便的部署在不同的硬件平台上，接下来我们首先将 YOLO v8 模型转换为 OpenVINO IR 模型格式。YOLOv8 提供了用于将模型导出到不同格式（包括 OpenVINO IR 格式）的 API。model.export 负责模型转换。我们需要在这里指定格式，此外，我们还可以在模型中保留动态输入。

```
from pathlib import Path

model_path = Path(f"{MODEL_NAME}_openvino_model/{MODEL_NAME}.xml")
if not model_path.exists():
    model.export(format="openvino", dynamic=True, half=False)
```

接下来我们来测试一下转换后模型的准确度如何。运行以下代码，并定义相应的前处理、后处理函数，

```
from openvino.runtime import Core, Model

core = Core()
ov_model = core.read_model(model_path)
device = "CPU"  # GPU
if device != "CPU":
    ov_model.reshape({0: [1, 3, 640, 640]})
compiled_model = core.compile_model(ov_model, device)
```

在单张测试图片上进行推理，可以得到如下推理结果



第三步： 在数据集上验证模型准确度

YOLOv8 是在 COCO 数据集上进行预训练的，因此为了评估模型的准确性，我们需要下载该数据集。根据 YOLOv8 GitHub 仓库中提供的说明，我们还需要下载模型作者使用的格式的标注，以便与原始模型评估功能一起使用。

```
import sys
from zipfile import ZipFile

sys.path.append("../utils")
from notebook_utils import download_file

DATA_URL = "http://images.cocodataset.org/zips/val2017.zip"
LABELS_URL = "https://github.com/ultralytics/yolov5/releases/download/v1.0/coco2017labels-segments.zip"

OUT_DIR = Path('./datasets')

download_file(DATA_URL, directory=OUT_DIR, show_progress=True)
download_file(LABELS_URL, directory=OUT_DIR, show_progress=True)

if not (OUT_DIR / "coco/labels").exists():
    with ZipFile(OUT_DIR / 'coco2017labels-segments.zip' , "r") as zip_ref:
        zip_ref.extractall(OUT_DIR)
    with ZipFile(OUT_DIR / 'val2017.zip' , "r") as zip_ref:
        zip_ref.extractall(OUT_DIR / 'coco/images')
```

接下来，我们配置 DetectionValidator 并创建 DataLoader。原始模型存储库使用 DetectionValidator 包装器，它表示精度验证的过程。它创建 DataLoader 和评估标准，并更新 DataLoader 生成的每个数据批的度量标准。此外，它还负责数据预处理和结果后处理。对于类初始化，应提供配置。我们将使用默认设置，但可以用一些参数替代，以测试自定义数据，代码如下。

```
from ultralytics.yolo.utils import DEFAULT_CONFIG
from ultralytics.yolo.configs import get_config
args = get_config(config=DEFAULT_CONFIG)
args.data = "coco.yml"
```

```
validator = model.ValidatorClass(args)

data_loader = validator.get_dataloader("datasets/coco", 1)
```

Validator 配置代码如下

```
from tqdm.notebook import tqdm
from ultralytics.yolo.utils.metrics import ConfusionMatrix

validator.is_coco = True
validator.class_map = ops.coco80_to_coco91_class()
validator.names = model.model.names
validator.metrics.names = validator.names
validator.nc = model.model.model[-1].nc
```

定义验证函数，以及打印相应测试结果的函数，结果如下

```
print_stats(fp_stats, validator.seen, validator.nt_per_class.sum())
```

```
               Class     Images     Labels  Precision     Recall     mAP@.5 mAP@.5:.95
                 all       5000      36335      0.633      0.474      0.521      0.371
```

第四步： 利用 NNCF POT 量化 API 进行模型优化

Neural network compression framework (NNCF) 为 OpenVINO 中的神经网络推理优化提供了一套先进的算法，精度下降最小。我们将在后训练（Post-training）模式中使用 8 位量化（无需微调）来优化 YOLOv8。

优化过程包括以下三个步骤：

1) 建立量化数据集 Dataset;
2) 运行 `nncf.quantize` 来得到优化模型
3) 使用串行化函数 `openvino.runtime.serialize` 来得到 OpenVINO IR 模型。

建立量化数据集代码如下

```
import nncf  # noqa: F811
from typing import Dict


def transform_fn(data_item:Dict):
    """
    Quantization transform function. Extracts and preprocess input data from dataloader item for quantization.
    Parameters:
       data_item: Dict with data item produced by DataLoader during iteration
    Returns:
        input_tensor: Input data for quantization
    """
    input_tensor = validator.preprocess(data_item)['img'].numpy()
```

```
    return input_tensor


quantization_dataset = nncf.Dataset(data_loader, transform_fn)
```

运行 `nncf.quantize` 代码如下

```
quantized_model = nncf.quantize(
    ov_model,
    quantization_dataset,
    preset=nncf.QuantizationPreset.MIXED,
    ignored_scope=nncf.IgnoredScope(
        types=["Multiply", "Subtract", "Sigmoid"],  # ignore operations
        names=["/model.22/dfl/conv/Conv",           # in the post-processing subgraph
               "/model.22/Add",
               "/model.22/Add_1",
               "/model.22/Add_2",
               "/model.22/Add_3",
               "/model.22/Add_4",
               "/model.22/Add_5",
               "/model.22/Add_6",
               "/model.22/Add_7",
               "/model.22/Add_8",
               "/model.22/Add_9",
               "/model.22/Add_10"]
    ))
```

最终串行化函数代码如下

```
from openvino.runtime import serialize
int8_model_path = Path(f'{MODEL_NAME}_openvino_int8_model/{MODEL_NAME}.xml')
print(f"Quantized model will be saved to {int8_model_path}")
serialize(quantized_model, str(int8_model_path))
```

运行后得到的优化的 YOLOv8 模型保存在以下路径

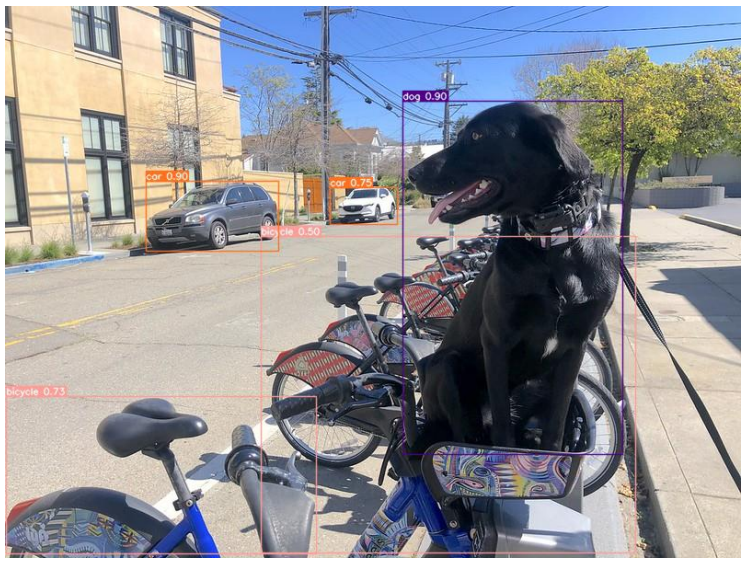
`yolov8n_openvino_int8_model/yolov8n.xml`

接下来，运行以下代码在单张测试图片上验证优化模型的推理结果

```
if device != "CPU":
    quantized_model.reshape({0, [1, 3, 640, 640]})
quantized_compiled_model = core.compile_model(quantized_model, device)
```

```
4. input_image = np.array(Image.open(IMAGE_PATH))
5. detections = detect(input_image, quantized_compiled_model)[0]
6. image_with_boxes = draw_boxes(detections, input_image)
7.
8. Image.fromarray(image_with_boxes)
```

运行结果如下



验证下优化后模型的精度，运行如下代码：

```
1. print("FP32 model accuracy")
2. print_stats(fp_stats, validator.seen, validator.nt_per_class.sum())
3.
4. print("INT8 model accuracy")
5. print_stats(int8_stats, validator.seen, validator.nt_per_class.sum()
   )
```

得到结果如下：

```
FP32 model accuracy
               Class      Images      Labels   Precision      Recall      mAP@.5  mAP@.5:.95
                 all        5000       36335       0.633       0.474       0.521       0.371
INT8 model accuracy
               Class      Images      Labels   Precision      Recall      mAP@.5  mAP@.5:.95
                 all        5000       36335       0.634       0.473       0.519       0.369
```

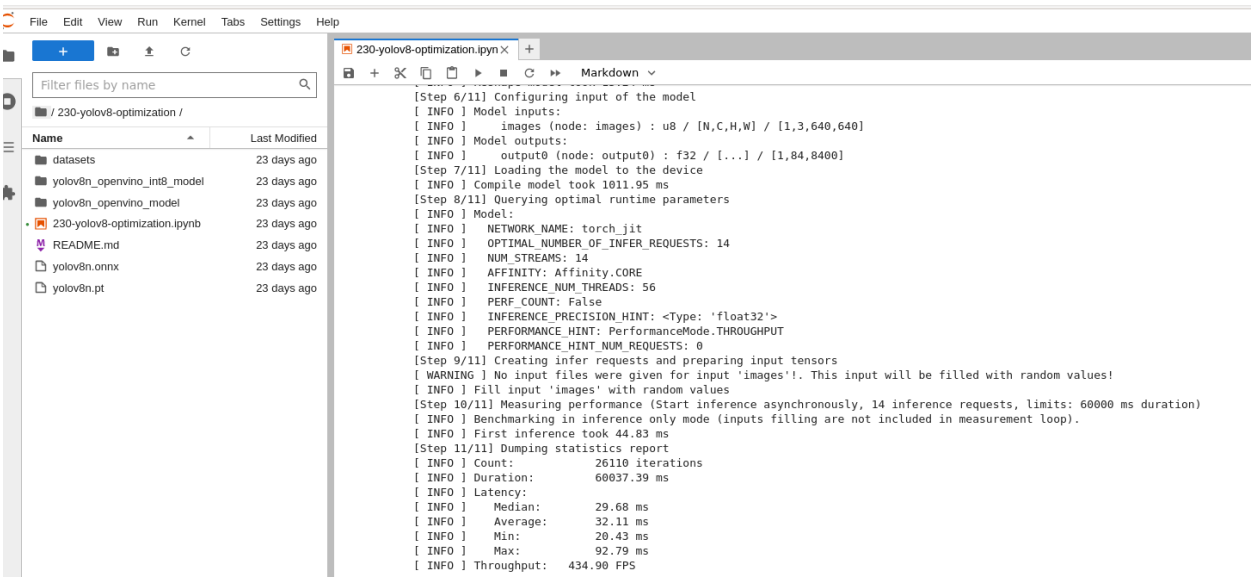
可以看到模型精度相较于优化前，并没有明显的下降。

第五步： 比较优化前后模型的性能

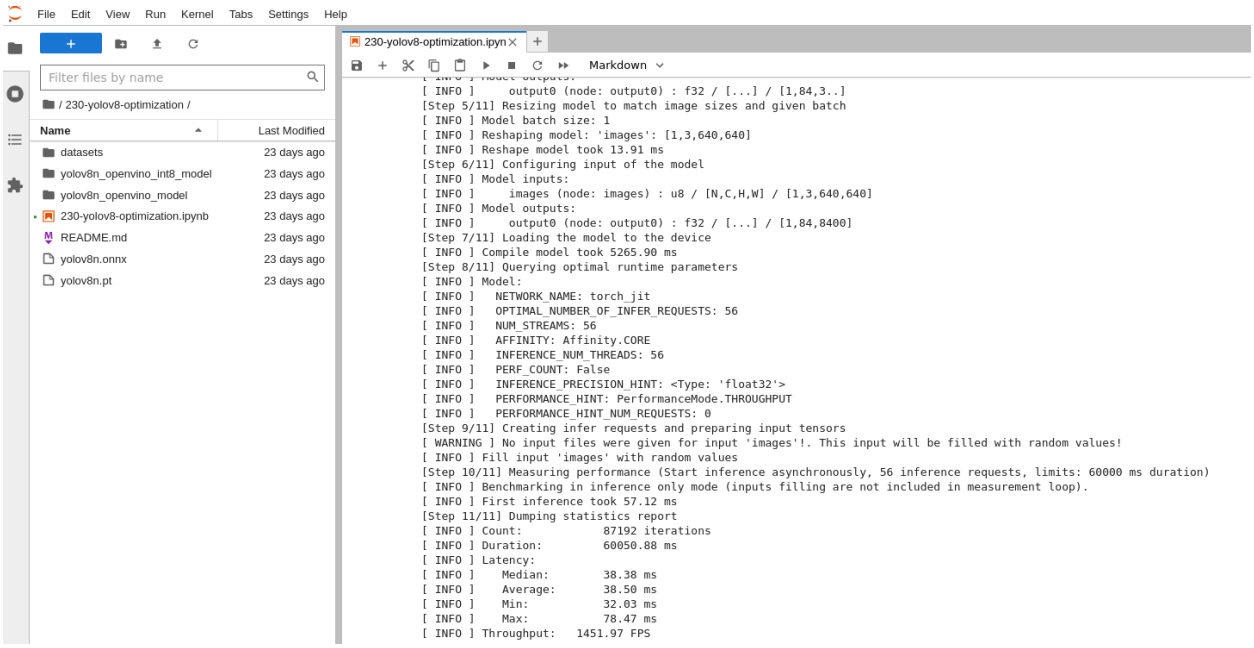
接着，我们利用 OpenVINO 基线测试工具 https://docs.openvino.ai/latest/openvino_inference_engine_tools_benchmark_tool_README.html 来比较优化前（FP32）和优化后（INT8）模型的性能。在这里，我们分别在英特尔®至强®第三代处理器（Xeon Ice Lake Gold Intel 6348 2.6 GHz 42 MB 235W 28 cores）上运行 CPU 端的性能比较。针对优化前模型的测试代码和运行结果如下

```
# Inference FP32 model (OpenVINO IR)
!benchmark_app -m $model_path -d CPU -api async -shape "[1,3,640,640]"
```

FP32 模型性能：

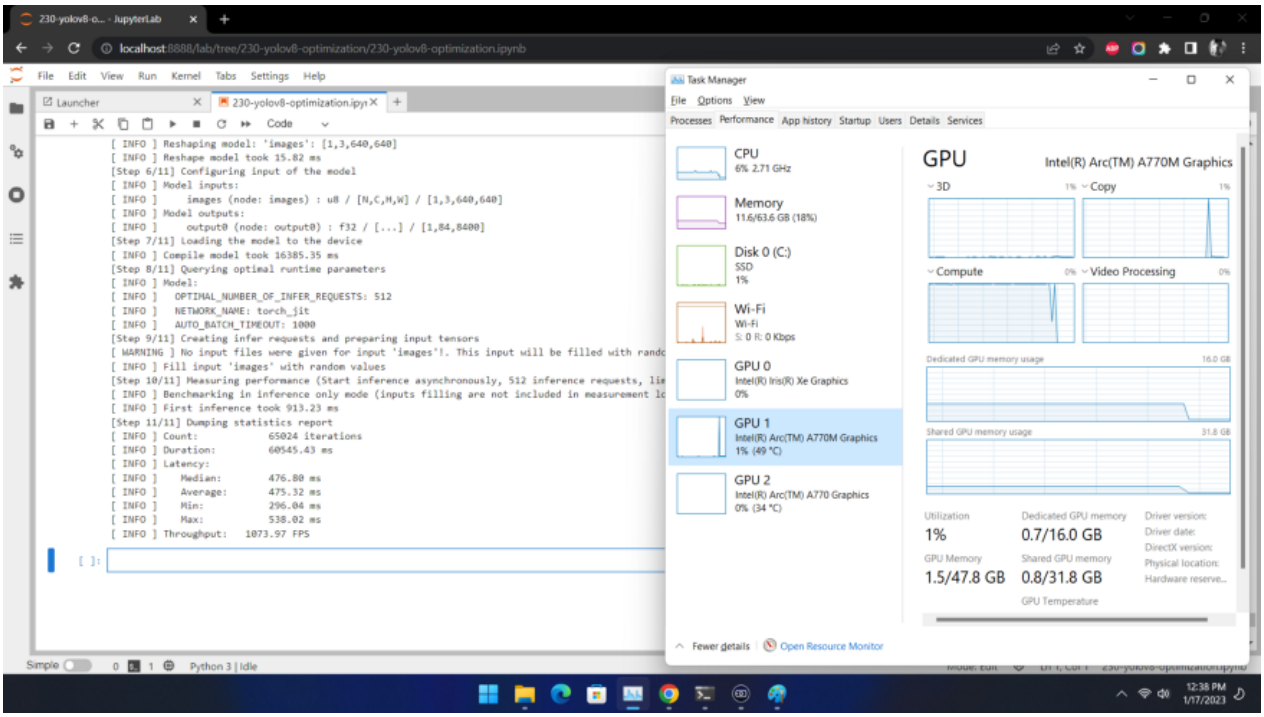


INT8 模型性能：



已经达到了 1400+ FPS!

在英特尔®独立显卡上的性能又如何呢？我们在 Arc™ A770m 上测试效果如下：



也超过了 1000 FPS!

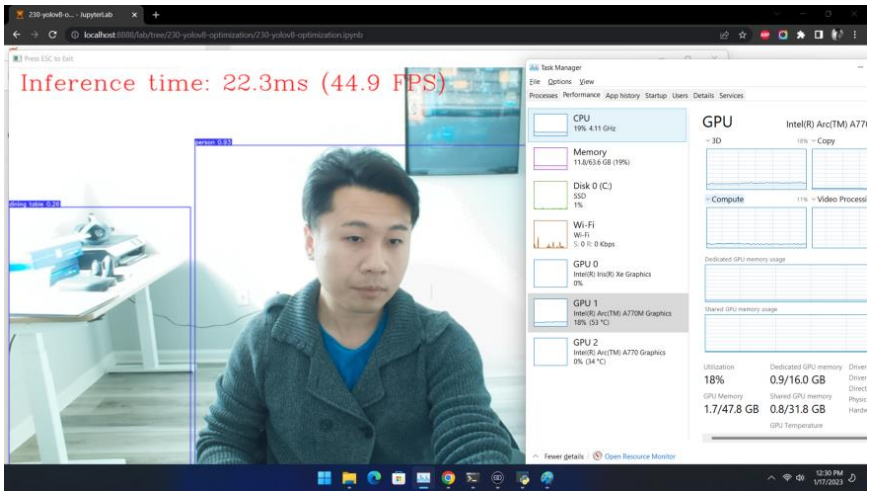
需要注意的是要想获得如此的高性能，需要将推理运行在吞吐量模式下，并使用多流和多个推理请求（即并行运行多个）。同样，仍然需要确保对预处理和后处理管道进行微调，以确保没有性能瓶颈。

第六步： 利用网络摄像头运行实时测试

除了基线测试工具外，如果你想利用自己的网络摄像头，体验一下实时推理的效果，可以运行我们提供的实时运行目标检测函数

```
1. run_object_detection(source=0, flip=True, use_popup=False, model=ov_
   model, device="AUTO")
```

获得类似如下图的效果：



第七步： 进一步提升性能的小技巧

- **非同步推理流水线**
  在进行目标检测的推理时，推理性能常常会因为数据输入量的限制而受到影响。此时，采用异步推理的模型，可以进一步提升推理的性能。异步 API 的主要优点是，当设备忙于推理时，应用程序可以并行执行其他任务（例如填充输入或调度其他请求），而不是等待当前推理首先完成。要了解如何使用 openvino 执行异步推理，请参阅 AsyncAPI 教程 https://github.com/openvinotoolkit/openvino_notebooks/blob/97f25b16970b6fe2287ca47bba64f31cff98e795/notebooks/115-async-api/115-async-api.ipynb 。

- **使用预处理 API**
  预处理 API 允许将预处理作为模型的一部分，从而减少应用程序代码和对其他图像处理库的依赖。预处理 API 的主要优点是将预处理步骤集成到执行图中，并将在选定的设备

（CPU/GPU/VPU/等）上执行，而不是作为应用程序的一部分始终在 CPU 上执行。这将提高所选设备的利用率。更详细的预处理 API 信息，请参阅预处理教程 https://docs.openvino.ai/latest/openvino_docs_OV_Runtime_UG_Preprocessing_Overview.html 。

对于本次 YOLOv8 示例来说，预处理 API 的使用包含以下几个步骤：

1. 初始化 PrePostProcessing 对象

```
20.from openvino.preprocess import PrePostProcessor
21.
22.ppp = PrePostProcessor(quantized_model)
```

2. 定义输入数据格式

```
2. from openvino.runtime import Type, Layout
3.
4. ppp.input(0).tensor().set_shape([1, 640, 640, 3]).set_element_type(Type.u8
   ).set_layout(Layout('NHWC'))
5. pass
```

3. 描述预处理步骤
   预处理步骤主要包括以下三步：
   - 将数据类型从 U8 转换为 FP32
   - 将数据布局从 NHWC 转换为 NCHW 格式
   - 通过按比例因子 255 进行除法来归一化每个像素

   代码如下：

```
1. ppp.input(0).preprocess().convert_element_type(Type.f32).convert_lay
   out(Layout('NCHW')).scale([255., 255., 255.])
2.
3. print(ppp)
```

4. 将步骤集成到模型中

```
1. quantized_model_with_preprocess = ppp.build()
2. serialize(quantized_model_with_preprocess, str(int8_model_path.with_
   name(f"{MODEL_NAME}_with_preprocess.xml")))
```

具有集成预处理的模型已准备好加载到设备。现在，我们可以跳过检测函数中的这些预处理步骤，直接运行如下推理

```
1. def detect_without_preprocess(image:np.ndarray, model:Model):
2.     """
```

```
    OpenVINO YOLOv8 model with integrated preprocessing inference function. Preprocess image, runs model inference and postprocess results using NMS.
    Parameters:
        image (np.ndarray): input image.
        model (Model): OpenVINO compiled model.
    Returns:
        detections (np.ndarray): detected boxes in format [x1, y1, x2, y2, score, label]
    """
    output_layer = model.output(0)
    img = letterbox(image)[0]
    input_tensor = np.expand_dims(img, 0)
    input_hw = img.shape[:2]
    result = model(input_tensor)[output_layer]
    detections = postprocess(result, input_hw, image)
    return detections


compiled_model = core.compile_model(quantized_model_with_preprocess, device)
input_image = np.array(Image.open(IMAGE_PATH))
detections = detect_without_preprocess(input_image, compiled_model)[0]
image_with_boxes = draw_boxes(detections, input_image)

Image.fromarray(img_with_boxes)
```

小结：

整个的步骤就是这样！现在就开始跟着我们提供的代码和步骤，动手试试用 Open VINO™优化和加速 YOLOv8 吧。

关于英特尔 OpenVINO™ 开源工具套件的详细资料，包括其中我们提供的三百多个经验证并优化的预训练模型的详细资料，请您点击 https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html

除此之外，为了方便大家了解并快速掌握 OpenVINO™ 的使用，我们还提供了一系列开源的 Jupyter notebook demo。运行这些 notebook，就能快速了解在不同场景下如何利用 OpenVINO™ 实现一系列、包括计算机视觉、语音及自然语言处理任务。OpenVINO™ notebooks 的资源可以在 Github 这里下载安装：https://github.com/openvinotoolkit/openvino_notebooks 。

通知和免责声明

Intel 技术可能需要启用硬件、软件或服务激活。

任何产品或组件都不能绝对安全。

您的成本和结果可能有所不同。