目　录

将数据预处理嵌入 AI 模型的常见技巧

作者：战鹏州 英特尔物联网行业创新大使

本文将介绍基于 OpenVINO™ 模型优化器或预处理 API 将数据预处理嵌入 AI 模型的常见技巧，帮助读者在硬件投入不变的情况下，进一步提升端到端的 AI 推理程序的性能。本文所有范例程序已开源：https://gitee.com/ppov-nuc/resnet_ov_ppp.git，并在基于第 12 代英特尔® 酷睿™ 处理器的 AI 开发者套件上完成测试。

以 YOLOv5 模型转换为例，使用模型优化器命令：

```
mo --input_model yolov5s.onnx --data_type FP16
```

将 yolov5s.onnx 模型转为 IR 格式模型，并将模型精度从 FP32 转为 FP16——这是最常见的模型优化器使用方式。基于上述 IR 模型，在编写 AI 推理程序时，由于图像数据的数值精度和形状，跟模型输入节点要求的数值精度和形状不一样，所以还需要将数据在输入模型前对其进行预处理。

以 YOLOv5 模型为例，使用 YOLOv5 代码仓自带的 zidane.jpg 图像，打印出图像的数值精度和形状，以及模型输入节点的数值精度和形状，对比如下，如图 1-1 所示。

```
import cv2
img = cv2.imread("./data/images/zidane.jpg")
print(img.shape,img[300,300,...],img.dtype)

from openvino.runtime import Core, get_version
core = Core()
net = core.compile_model("yolov5s.xml","CPU")
print(net.inputs[0])
```

✓ 0.2s

```
(720, 1280, 3) [ 63  83 131] uint8
<ConstOutput: names[images] shape{1,3,640,640} type: f32>
```

图 1-1 OpenCV 读入的图像 vs 模型输入节点

从上图可以看出，通过 OpenCV 的 imread()函数读取的图像数据，在数据形状、数值精度、数值范围等地方，与模型输入节点的要求不一样，如下表所示

	OpenCV 读取的图像数据	YOLOv5 模型输入节点
数据形状(Shape)	[720, 1280, 3]	[1, 3, 640, 640]
数值精度(dtype)	UINT8	FP32
数值范围	0 - 255	0.0 - 1.0
颜色通道顺序	BGR	RGB
数据布局(Layout)	HWC	NCHW

由于存在上述的差异，数据在传入模型前必须进行预处理，以满足模型输入节点的要求。数据预处理可以在推理代码中编程实现，也可以用模型优化器实现，或者用 OpenVINO™ 预处理 API 实现，本文将依次详细介绍。

1.1 用模型优化器实现数据预处理

1.1.1 模型优化器预处理参数

模型优化器可以将颜色通道顺序调整、图像数据归一化等预处理操作嵌入模型，参考《OpenVINO™ 模型转换技术要点解读》。通过指定参数：

- --mean_values：所有输入数据将减去 mean_values，即 input - mean_values
- --scale_values：所有输入数据将除以 scales_values，当同时指定 mean_values 和 scale_values 时，模型优化器执行(input － mean_values)÷scales_values
- --reverse_input_channels：将输入通道顺序从 RGB 转换为 BGR（反之亦然）

当上述三个操作同时指定时，预处理顺序为：

```
输入数据→reverse_input_channels→mean_values→scale_values→原始模型
```

在转换模型时，假设推理程序使用 OpenCV 库读取图像，则可以在模型优化器中增加 mean_values、scale_values 和 reverse_input_channels 三个参数，把颜色通道顺序调整和图像数据归一化操作嵌入模型。若推理程序使用非 OpenCV 库读取图像，例如 PIL.Image，则无需添加--reverse_input_channels 参数。

下面本文将以 ResNet 模型为例，展示使用模型优化器将预处理嵌入模型的完整过程。

1.1.2 将 ResNet 模型的预处理嵌入模型

ResNet 不仅是 2015 年 ILSVRC 大赛冠军，还是产业实践中常用的卷积神经网络模型。PyTorch 已将 ResNet 集成到 torchvision 中，将 PyTorch 格式的 ResNet 模型转为 ONNX 格式，完整代码如下：

```
from torchvision.models import resnet50, ResNet50_Weights
import torch
# https://pytorch.org/vision/stable/models/generated/torchvision.models.resnet50.html
weights = ResNet50_Weights.IMAGENET1K_V2
model = resnet50(weights=weights, progress=False).cpu().eval()
# define input and output node
dummy_input = torch.randn(1, 3, 224, 224, device="cpu")
input_names,  output_names = ["images"], ['output']
torch.onnx.export(model,
                 dummy_input,
                 "resnet50.onnx",
                 verbose=True,
                 input_names=input_names,
                 output_names=output_names,
                 opset_version=13
                 )
```

在导出 PyTorch 格式模型为 ONNX 格式时，需要注意的是算子版本(opset_version)最好≥11。另外，OpenVINO2022.2 支持 ONNX 1.8.1，即 opset_version=13，所以本文将 opset_version 设置为 13。

基于 ImageNet 1k 数据集训练的 ResNet 模型的归一化参数为：

- mean_values= [123.675,116.28,103.53]
- scale_values=[58.395,57.12,57.375]

将 ONNX 模型转换为 OpenVINO™ IR 模型的命令为：

```
mo -m resnet50.onnx --mean_values=[123.675,116.28,103.53] --
scale_values=[58.395,57.12,57.375]  --data_type FP16 --reverse_input_channels
```

当获得 ResNet50 的 IR 模型后，可以使用下面的程序，完成推理计算

```
from openvino.runtime import Core
import cv2
import numpy as np
core = Core()
resnet50 = core.compile_model("resnet50.xml", "CPU")
output_node = resnet50.outputs[0]
# Resize
img = cv2.resize(cv2.imread("cat.jpg"), [224,224])
# Layout: HWC -> NCHW
blob = np.expand_dims(np.transpose(img, (2,0,1)), 0)
result = resnet50(blob)[output_node]
print(np.argmax(result))
```

在上面的推理代码中，仍有调整图像尺寸，改变图像数据布局等操作在推理代码中实现，接下来，本文将介绍用 OpenVINO™ 预处理 API，将更多预处理操作嵌入模型中。

1.2 用 OpenVINO™ 预处理 API 实现数据预处理

从 OpenVINO™ 2022.1 开始，OpenVINO 提供一套预处理 API，将数据预处理嵌入模型，参考《使用 OpenVINO™ 预处理 API 进一步提升 YOLOv5 推理性能》。将数据预处理嵌入模型的好处是：

- ✓ 提高 AI 模型的移植性(推理代码无需考虑编写预处理程序)
- ✓ 提高推理设备(例如，英特尔®集成显卡/独立显卡)的利用率
- ✓ 提高 AI 程序端到端的性能

使用 OpenVINO™ 预处理 API 将预处理嵌入模型的完整范例程序 export_resnet_ov_ppp.py，如下所示：

```
from openvino.preprocess import PrePostProcessor, ColorFormat, ResizeAlgorithm
from openvino.runtime import Core, Layout, Type, serialize
# ========  Step 0: read original model =========
```

```python
core = Core()

model = core.read_model("resnet50.onnx")

# ======== Step 1: Preprocessing ================

ppp = PrePostProcessor(model)

# Declare section of desired application's input format

ppp.input("images").tensor() \

    .set_element_type(Type.u8) \

    .set_spatial_dynamic_shape() \

    .set_layout(Layout('NHWC')) \

    .set_color_format(ColorFormat.BGR)

# Specify actual model layout

ppp.input("images").model().set_layout(Layout('NCHW'))

# Explicit preprocessing steps. Layout conversion will be done automatically as last step

ppp.input("images").preprocess() \

    .convert_element_type()     \

    .convert_color(ColorFormat.RGB) \

    .resize(ResizeAlgorithm.RESIZE_LINEAR) \

    .mean([123.675, 116.28, 103.53]) \

    .scale([58.624, 57.12, 57.375])

# Dump preprocessor

print(f'Dump preprocessor: {ppp}')

model = ppp.build()

# ======== Step 2: Save the model with preprocessor================

serialize(model, 'resnet50_ppp.xml', 'resnet50_ppp.bin')
```

export_resnet_ov_ppp.py 运行结果，如下图所示：

```
(ppov) C:\Users\NUC\Desktop\models\ResNet>python export_resnet_ov_ppp.py
Dump preprocessor: Input "images" (color BGR):
    User's input tensor: {1,?,?,3}, [N,H,W,C], u8
    Model's expected tensor: {1,3,224,224}, [N,C,H,W], f32
    Pre-processing steps (5):
      convert type (undefined): ({1,?,?,3}, [N,H,W,C], u8, BGR) -> ({1,?,?,3}, [N,H,W,C], f32, BGR)
      convert color (RGB): ({1,?,?,3}, [N,H,W,C], f32, BGR) -> ({1,?,?,3}, [N,H,W,C], f32, RGB)
      resize to model width/height: ({1,?,?,3}, [N,H,W,C], f32, RGB) -> ({1,224,224,3}, [N,H,W,C], f32, RGB)
      mean (123.675,116.28,103.53): ({1,224,224,3}, [N,H,W,C], f32, RGB) -> ({1,224,224,3}, [N,H,W,C], f32, RGB)
      scale (58.624,57.12,57.375): ({1,224,224,3}, [N,H,W,C], f32, RGB) -> ({1,224,224,3}, [N,H,W,C], f32, RGB)
    Implicit pre-processing steps (1):
      convert layout [N,C,H,W]: ({1,224,224,3}, [N,H,W,C], f32, RGB) -> ({1,3,224,224}, [N,C,H,W], f32, RGB)
```

从上面的代码可见，使用 OpenVINO™ 预处理 API，可以将图像尺寸调整、彩色通道转换、数据归一化、数据布局转换全部集成到模型中，并且无需运行模型优化器，即可以将 ONNX 模型导出为 IR 模型。

基于 resnet50_ppp.xml 的完整推理程序，如下所示：

```python
from openvino.runtime import Core

import cv2
```

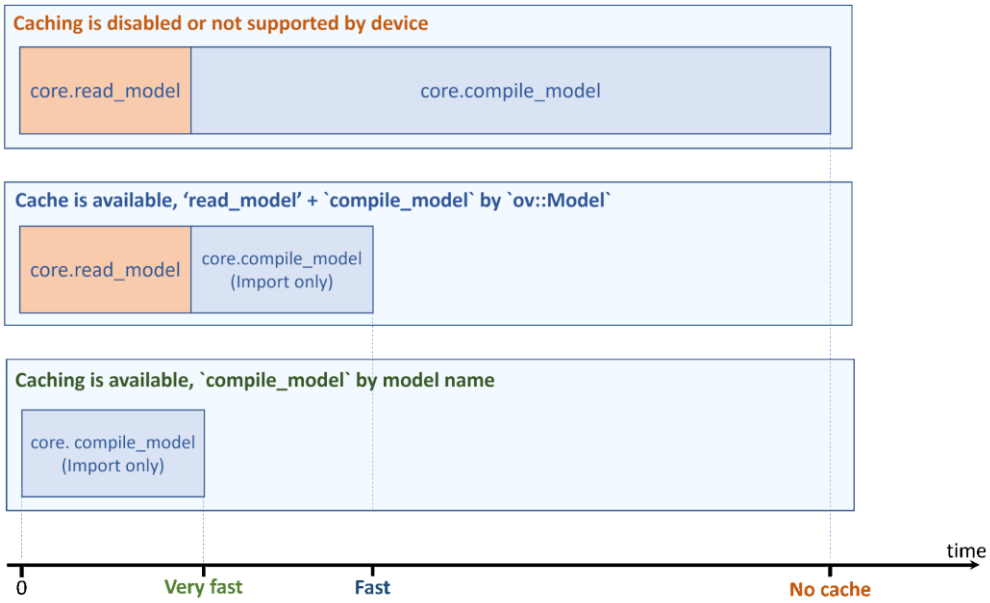
```
import numpy as np

core = Core()

resnet50_ppp = core.compile_model("resnet50_ppp.xml", "CPU")

output_node = resnet50_ppp.outputs[0]

blob = np.expand_dims(cv2.imread("cat.jpg"),0)

result = resnet50_ppp(blob)[output_node]

print(np.argmax(result))
```

如上所示，基于内嵌预处理的 IR 模型，OpenVINO 推理程序变得更加简单清晰，易读易懂了。**五行 Python 核心代码实现了内嵌预处理的 ResNet 模型推理！**

1.3 使用模型缓存技术进一步缩短首次推理时延

在《在蝰蛇峡谷上实现 YOLOv5 模型的 OpenVINO 异步推理程序》讨论了 AI 应用程序端到端的性能。对于首次推理时延来说，模型的载入和编译时间，会极大增加首次推理的端到端的运行时间。

使用模型缓存技术，将极大的缩短首次推理时延，如下图所示。



使用模型缓存技术，只需要添加一行代码：core.set_property({'CACHE_DIR': './cache/ppp'})，完整范例代码如下所示：

```
from openvino.runtime import Core

import cv2

import numpy as np

core = Core()

core.set_property({'CACHE_DIR': './cache/ppp'}) # 使用模型缓存技术

resnet50_ppp = core.compile_model("resnet50_ppp.xml", "CPU")

output_node = resnet50_ppp.outputs[0]

blob = np.expand_dims(cv2.imread("cat.jpg"),0)

result = resnet50_ppp(blob)[output_node]
```

```
print(np.argmax(result))
```

当第二次运行推理程序时，OpenVINO™ 运行时将从缓存文件夹直接加载已编译好的模型，极大的优化了首次推理时延。

1.4 总结

本文详细介绍了通过模型优化器和 OpenVINO™ 预处理 API，将数据预处理嵌入 AI 模型的技术。将数据预处理嵌入模型，简化了推理程序编写，提升推理计算设备利用率并提升了 AI 程序端到端的性能。最后，本文还介绍了通过模型缓存技术，进一步优化 AI 程序端到端的首次推理时延性能。