当前项目，主要基于当前比较常见的OpenVINO™、TensorRT、ONNX runtime 以及 OpenCV Dnn 四个部署平台进行对比展开，主要针对这四个平台如何部署深度学习项目进行了测试，并针对OpenVINOTM、TensorRT 提出了在 C#编程平台的解决办法，建立了 C#模型部署检测软件平台。

AI 模型部署平台

不同部署框架在 C++与 C#部署 AI 模型实现

颜国进

2022 年 6 月

项目说明

该项目目前已经实现了 OpenVINO™、TensorRT、ONNX runtime 以及 OpenCV Dnn 四种模型部署工具在 C++、C#平台使用，其中由于 OpenVINO™、TensorRT 未提供 C#语言接口，该项目中提出了解决方法。

由于笔者水平有限且时间紧促，因此在写文档时会有错误地方，后续使用者在使用时如有问题可以发邮件咨询(guojin_yjs@cumt.edu.cn)或者在 GitHub 以及 Gitee 相应项目下留言。并且欢迎大家对该项目提出意见，后续会相应改正。

项目文档编写不易，欢迎大家下载学习与使用，推动在 C#平台使用 OpenVINO™、TensorRT 等工具进行模型部署。未经作者同意严禁将该文档内容上传其他平台进行盈利行为。

第1章 项目概述

1.1 人工智能项目

人工智能，也就是我们常说的“AI”（英文全称：Artificial Intelligence），是研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。它是以人类智能相似的方式做出反应的智能机器，领域研究包括机器人、语言识别、图像识别、自然语言处理和专家系统等。

人工智能自诞生以来，理论和技术日益成熟，应用领域也不断扩大。可以设想，未来人工智能带来的科技产品，将会是人类智慧的“容器”。人工智能可以对人的意识、思维的信息过程的模拟，虽然不是人的智能，但能像人那样思考，也可能超过人的智能。

近几年来，随着芯片算力的不断提升和数据的不断增长，深度学习算法有了长足的发展。深度学习算法也越来越多的应用在各个领域中，比如图像处理在安防领域和自动驾驶领域的应用，再比如语音处理和自然语言处理，以及各种各样的推荐算法。

1.2 人工智能项目基本步骤

经典的人工智能项目从提出到最后应用落地主要有以下几个步骤：

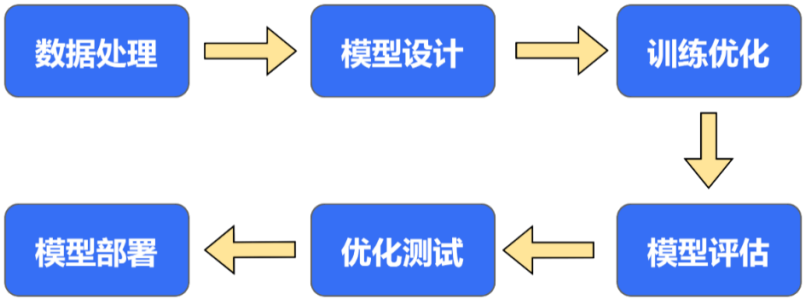


图 1- 1 人工智能项目基本步骤

1.2.1 数据处理

对于人工智能项目，第一步就是数据处理，这一步主要分为数据收集与数据预处理，人工智能其实现方式主要是通过大量的学习数据将训练网络培养成一个资深的专家，后面如果在有数据输入，这个专家就会根据前面学些的知识做出判断。这个判断到底准不准确，其主要有前面的学习来决定，这就和我们人一样，只有学习好了才能做出正确判断。因此数据处理这一步主要是要实现后续学习数据的准备，因此在这一步要求数据量要大，数据要正确。

数据收集这一步主要是针对我们所要做的项目，收集相关的数据。例如我要训练一个网络，这个网络可以识别猫、狗等动物，因此我要大量收集猫和狗等动物的大量照片，作为后续学习的数据。数据预处理这一块，主要是要标注出每一条数据正确的结果。在猫狗动物识别这个数据中，我需要做的就是告知其每张照片对应的动物是啥，这一步是比较费时间与精力的，因为他不能用电脑自动完成，只能人工一个个进行标注。除此以外，为了减少运算，增加网络学习速度，我们还会将数据进行归一化处理。

1.2.2 模型设计

模型设计也是比肩关键的一步，如果数据是材料，那么模型就是容器，好的材料配上好的丹炉，才有产出好的丹药的可能。模型的发展是深度学习技术发展中较为重要的一

环，模型的发展都朝着一个共同的目标前进，这个目标就是更高的精度、更高的训练效率、更快的推理速度、更大的平台适应性。这也是学术界重点关注和发展的方向，因为模型的轻微提升，就可以发表一篇论文，甚至可以发表在顶会期刊上面，因为学术研究的就是在固定条件下的模型性能和效率。而工程界的条件不可能像学术界那样固定，每个具体的项目要求都不一样的，而且不同项目的数据也是大相径庭，这样就会造成同样的模型，在不同条件下会出现不同的结果。对于工程界，其关注的最多的是人工智能的最后两步：测试调整与模型部署。

目前网络很多，而且发展很快，最开始的卷积神经网络模型 LeNet，还是后来的 AlexNet，以及后面陆续出现的 VggNet、GoogLeNet、SqueezeNet、ResNet、DenseNet、DarkNet、MobileNet、ShuffleNet、EfficientNet，又或者是这两年异军突起的 Transformer 模型，以及 CNN 和 Transformer 的结合体，网络也在不断升级中。

1.2.3 训练优化

针对模型设计这一步，我们目前应该还没有达到设计阶段，我们目前主要做的是选择合适的网络，带入我们的数据集进行学习。因此我们重中之重就是选择什么样的主干模型，进行怎么样的微调，以及选择什么样的损失函数和优化方法，是否进行多阶段训练，或者对图像数据进行多尺度训练等。

一般来说，有经验的训练人员，在训练模型的时候，不会一开始就输入大量的数据，而是做小批量的数据的训练，目的是先通过小批量数据来检验模型的好坏，然后再根据检验情况进行下一步的操作。因为对于数据集很大的训练，可能训练一次就要好几天，是十分耽误时间的。目前比较合适的做法是，在般模型训练开始之前先使用少量的数据快速的训练出一个过拟合的结果，用来验证模型是否合适，过拟合往往意味着参数过大，或者数据过少，导致模型测试的时候泛化能力太差，但可以说明模型本身是没有问题的，问题就在于训练过度，或者数据集过少。后面的话再继续追加数据集的数量，实现网络的训练。

1.2.4 评估验证

模型评估是和模型训练伴随而行的，可以说训练一开始，评估也随之开始。模型训练只负责如何把模型训练好，至于要训练到什么程度才算合适，需要模型评估说了算。所以在开始一项任务的时候，模型训练和模型评估是同时进行的，正常情况下，模型训练一次，则评估一次，但是我们认为模型训练的前几次是不需要评估的，因为模型训练的前期，参数还没有学习到正确的数值。

根据经验，一般模型在训练到一定的次数之后，再启动模型验证部分，相比模型训练一开始就启动模型评估，这样的操作可以在保证模型有效评估的前提下加快模型训练速度，因为模型在训练到后期的时候，只凭经验是很难确定模型是否已经训练到合适的程度的。有了评估验证的过程，就可以根据验证的结果来判断模型是否需要继续训练了。

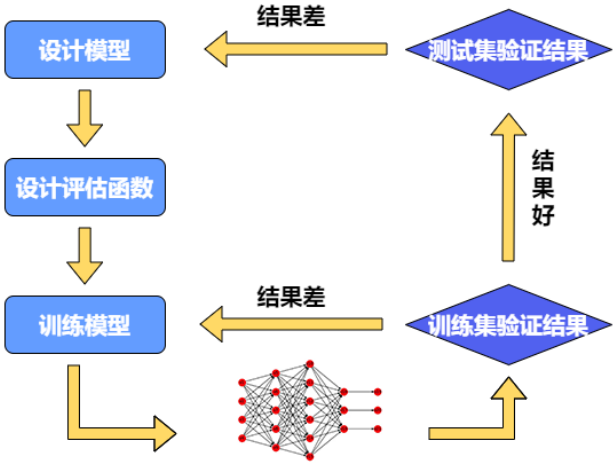


图 1- 2 模型训练与评估验证之间的关系

1.2.5 测试调整

模型测试是项目交付前的最后一次试验，测试的目的就是和项目方给出的指标做对比，比如精度、速度等指标。所以作为项目交付前的最后一次测试实验，一定要按照项目方提出的指标要求做测试，测试的数据最好是从项目方实际的工作场景中采集，如果条件不允许，那么测试数据一定要最大可能的接近项目方实际工作场景的数据，只有这样才有可能在项目交付后不会出现算法指标上的问题。

1.2.6 模型部署

对于一个完整的 AI 项目，前面这一些步骤可以说是在实验室完成的，但 AI 项目最终是要应用的具体的项目中，模型部署就是完成这个工作的。因此，一般来说，学术界负责各种 SOTA(State of the Art) 模型的训练和结构探索，而工业界负责将这些 SOTA 模型应用落地，赋能百业。模型部署一般无需再考虑如何修改训练方式或者修改网络结构以提高模型精度，更多的是需要明确部署的场景、部署方式（中心服务化还是本地终端部署）、模型的优化指标，以及如何提高吞吐率和减少延迟等。

人工智能经过多年的发展，不少也推出了自己的模型框架用于深度学习的训练，比较常见的有 Pytorch、TensorFlow、Caffe 以及 Paddle 飞桨等框架，这些平台在提升模型搭建效率、增加模型训练速度等方面，都做出了对应的提升。针对不同的训练框架平台，训练得到的模型文件也是不同的，目前各大厂商都将 ONNX 模型格式作为中间的转换格式，支持对该格式的转换，ONNX 是微软公司推出的一种模型文件格式，目前已经被大家所认可使用，向前来看，它支持各种训练框架转换，向后看，它也支持各种推理引擎的读取，如图 1- 3 所示。

近些年来，在深度学习算法已经足够卷之后，深度学习的另一个偏向于工程的方向—部署工业落地，才开始被谈论的多了起来。当然这也是大势所趋，毕竟 AI 算法那么多，如果用不着，只在学术圈搞研究的话没有意义。因此目前市面中的推理引擎平台，也是遍地开花，比较有代表的是英特尔推出的 OpenVINO™ 以及英伟达推出的 TensorRT 部署平台，这两大厂家都是是做芯片出身，因此他们基于自家芯片特性，推出了基于自家芯片的推理引擎。OpenVINO™ 是目前在 CPU 上运行最快的推理平台，TensorRT 是在 GPU 上运行最快的推理平台。除此以外，微软基于自己的 ONNX 模型，推出了 ONNX runtime 推理平台；OpenCV 作为当前最大的视觉处理开源库，也推出了自家的模型部署工具 OpenCV Dnn；除此以外还有其他的模型部署框架，但未在当前项目中涉及。

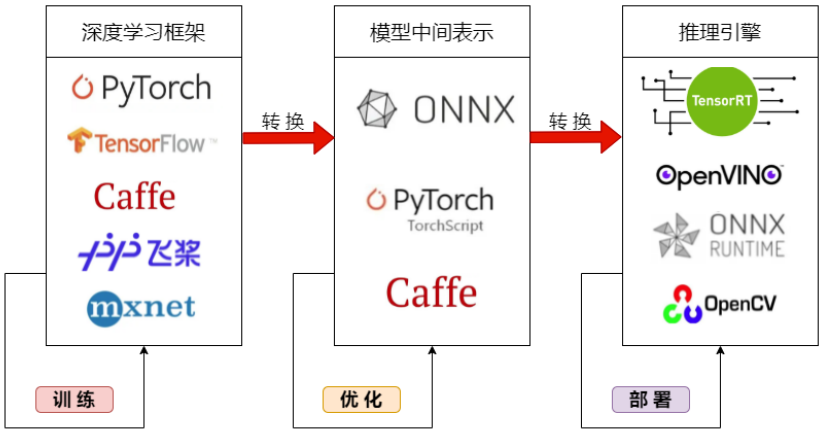


图 1- 3 常见深度学习框架、模型中间表示及推理引擎

1.3 模型部署

模型部署通俗地来说就是调用硬件算力，加速深度学习网络在新平台以及新的数据下进行推理。如图 1- 4 所示，模型在训练后，经过模型部署套件的优化与部署后，在硬件平台上进行模型推理。模型的训练阶段主要在实验室完成，为了获取高的推理精度，因此不会考虑模型的推理效率、速度以及内存大小，但在模型部署阶段，其主要是适应工业化的要求，因此需要极高的推理速度；并且对于一些终端设备，内存有限，所以也要考虑内存的限制，因此需要模型优化，以满足工业上的需求。

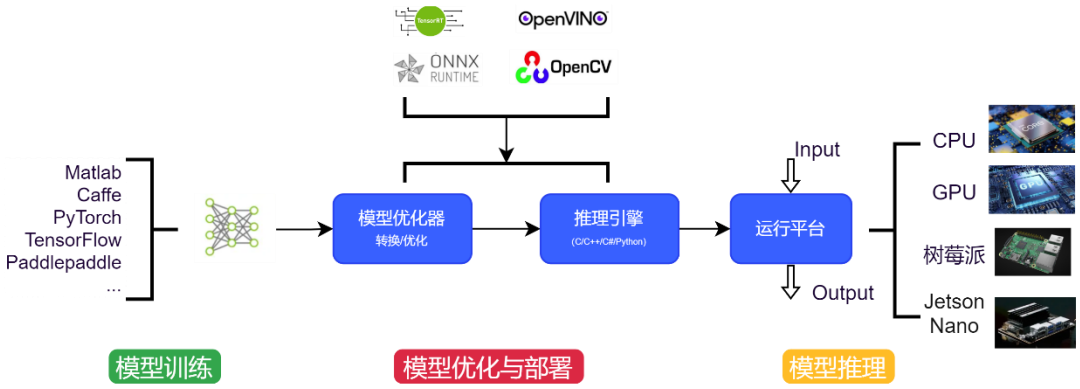


图 1- 4 推理模型部署流程

如何评价模型部署，主要有一个 AI 模型部署九字诀：跑起来、跑得快、持续跑。评价模型部署工具好不好，主要依据就是考虑是否符合上述要求；在选择部署工具时，也可以通过这三点进行选择。

对于跑起来，主要就是适配：硬件适配、软件（系统）适配以及框架适配。

硬件适配主要指的是部署套件与计算机硬件的适配关系，好的适配关系部署工具可以充分调用硬件算力进行模型推理，常见的芯片有 CPU、GPU、ARM 芯片等。目前 Intel 厂家基于自家生产的芯片，推出了 OpenVINO™，该套件再经过测试后，可以在 CPU 上表现出很好的性能；NVDIA 基于自家 GPU 推出的 TensorRT，该部署工具可以实现在 GPU 显卡部署推理模型，并且经过测试后可以看出推理速度有了很大的提升。目前这两大芯片厂

家基于自家芯片推出的推理工具，极大地发挥了各自芯片的水平。

软件（系统）适配主要是推理套件要支持各大软件系统，常见的系统有 Linux、Windows、Android 以及 iOS 系统，对于使用者来说，部署的位置对象是多样的，所以要考虑套件系统的适配性。框架适配指的是深度学习框架与推理工具支持的框架要一致；不过目前中间格式 ONNX 框架的对于模型前后的支持，使得不少推理套件更好的至此更多的模型。

对于跑得快来说，表面上来说就是推理速度要快，运行内存要低，但不能损失精度。这个一方面可以通过提升硬件算力以及并行处理能力来提高，不过目前各大硬件厂家也已经在努力提升；另一方面就是优化模型，目前对我们来说优化模型是可以实现的。模型优化的方法主要有剪枝、蒸馏、量化等方式，不过这些处理方式最总是要来不牺牲模型精度的前提下，提升模型的推理速度。

对于持续跑来说，就是在维持模型能够持续运行的基础上，并不断提升模型的精度御速度，优化模型结构。

1.4 AI 部署平台项目

1.4.1 项目介绍

当前项目，主要基于当前比较常见的 OpenVINO™、TensorRT、ONNX runtime 以及 OpenCV Dnn 四个部署平台进行对比展开，主要针对这四个平台如何部署深度学习项目进行了测试，并针对 OpenVINO™、TensorRT 提出了在 C#编程平台的解决办法，建立了 C#模型部署检测软件平台。

1.4.2 项目安装方式

在 Github 上克隆下载：

git clone https://github.com/guojin-yan/Inference.git

在 Gitee 上克隆下载：

git clone https://gitee.com/guojin-yan/Inference.git

第2章 软件平台准备

2.1 Microsoft Visual Studio 2022

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境(IDE)等等。其支持 C、C++、C#、F#、J#等多门编程语言。

本次项目所使用的编程语言为 C++与 C#两门编程语言，在 VS 中完全可以实现，可选择安装版本 VS2017、VS2019 或 VS2022 版本。对于 VS 不同版本的选择，该项目不做较多要求，就笔者使用来说，VS2017 版本推出时间较久，不建议使用，其一些编程语言规范有一些变动，对于该项目所提供的范例可能会有部分不兼容；VS2019 和 VS2022 版本相对更新，是由起来差异不大，建议选择这两个版本，并且新版 OpenVINO™ 支持 VS2022 版本 Cmake。

笔者电脑安装的为 Microsoft Visual Studio Community 2022 版本，其安装包可由 VS 官网直接下载，下载时选择社区版，按照一般安装步骤进行安装即可。在安装中，根据我们的使用的要求，工作负荷的选择如图 2- 1 所示。

图 2- 1 Visual Studio 2022 安装负荷

安装完成后，可以参照网上相关教程，进行学习 VS 的使用。如果在其他编辑器上使用，后续软件安装请根据编辑器要求进行配置，此处不做过多讲解，优先选用 VS2019 与 VS2022。

2.2 OpenVINO™

2.2.1 OpenVINO™ 介绍

OpenVINO™是英特尔基于自身现有的硬件平台开发的一种可以加快高性能计算机视觉和深度学习视觉应用开发速度工具套件，用于快速开发应用程序和解决方案，以解决各种任务（包括人类视觉模拟、自动语音识别、自然语言处理和推荐系统等）。

该工具套件基于最新一代的人工神经网络，包括卷积神经网络 (CNN)、递归网络和基于注意力的网络，可扩展跨英特尔® 硬件的计算机视觉和非视觉工作负载，从而最大限度地提高性能。它通过从边缘到云部署的高性能、人工智能和深度学习推理来为应用程序加

速，并且允许直接异构执行。

- 提高计算机视觉、自动语音识别、自然语言处理和其他常见任务中的深度学习性能
- 使用使用流行的框架（如 TensorFlow，PyTorch 等）训练的模型
- 减少资源需求，并在从边缘到云的一系列英特尔®平台上高效部署
- 支持在 Windows 与 Linux 系统，官方支持编程语言为 Python 与 C++语言。

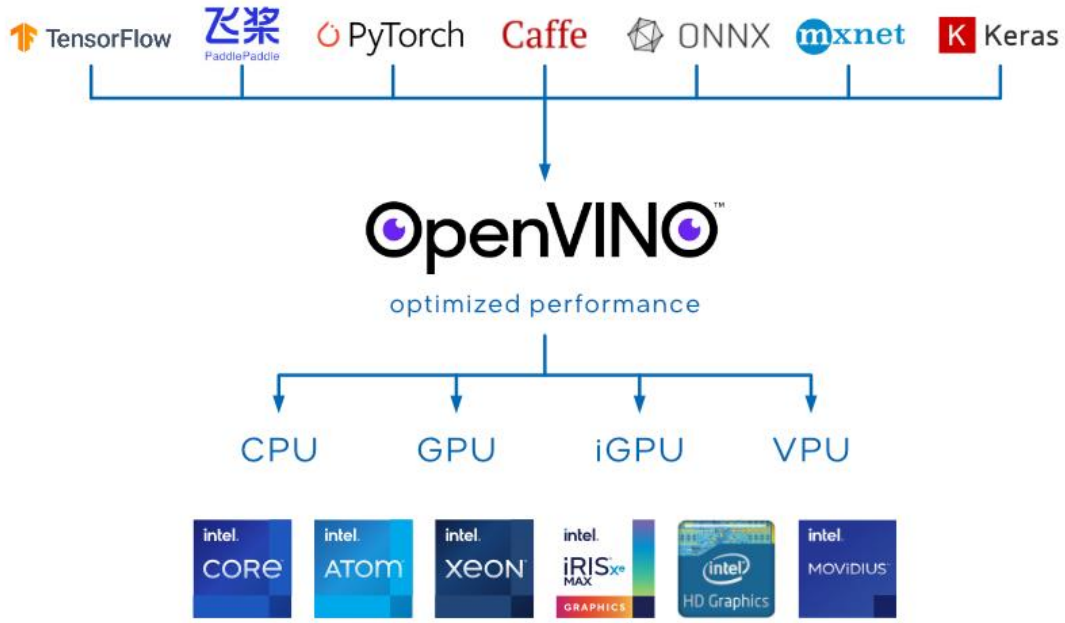


图 2- 2 OpenVINO™ 工具结构图

OpenVINO™ 工具套件 2022.1 版于 2022 年 3 月 22 日正式发布，根据官宣《OpenVINO™ 迎来迄今为止最重大更新，2022.1 新特性抢先看》，OpenVINO™ 2022.1 将是迄今为止最大变化的版本。从开发者的角度来看，对于提升开发效率或运行效率有用的特性有：

- 提供预处理 API 函数
- ONNX 前端 API
- AUTO 设备插件
- 支持直接读入飞桨模型

该项目所使用的 OpenVINO™ 版本为 2022.1 版本， OpenVINO™ 安装分为完整版安装与不完整版安装，对于在 C++中使用，选者非完整版安装即可，不过 OpenVINO™ 提供的相关工具不会安装，完整版安装主要通过 PIP 工具实现，最好可以将其安装在 Anaconda 虚拟环境下，因此此处只详细介绍非完整版安装。

2.2.2 OpenVINO™ 安装

（1）OpenVINO™ 下载

访问 OpenVINO™ 官网<www.openvino.ai>，点击 Free Download->，进入到软件下载页面，按照图 2- 3 所示，选择安装内容：

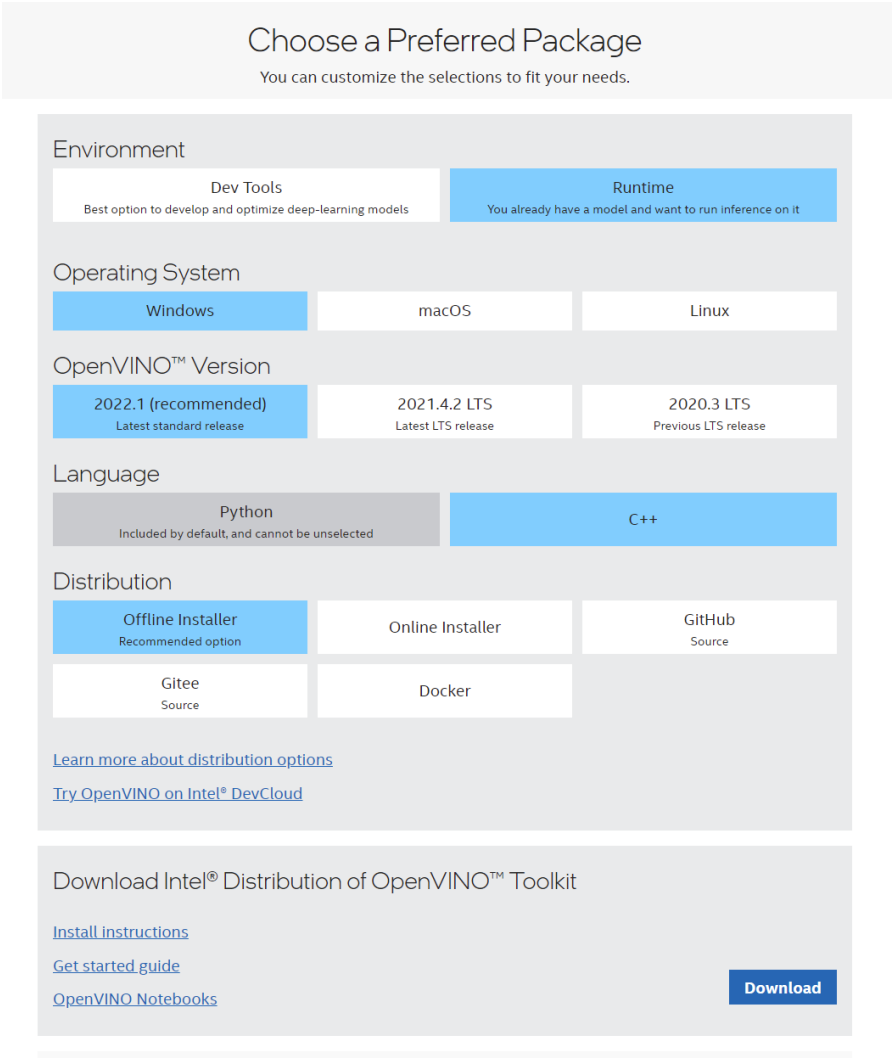


图 2- 3 OpenVINO™ 安装选择

选择完成后，点击 Download，下载安装包。

（2）安装软件

在安装包下载完成后，直接安装软件即可，全程默认软件安装即可，不需要做任何修改。

安装完成后，可以打开< C:\Program Files (x86)\Intel >路径，查看该文件夹下是否有< openvino_2022.1.0.643 >文件夹，若存在，说明安装成功。

（3）配置环境变量

在软件安装完成后，需要配置相关环境变量，防止每次使用都需要运行虚拟环境。右击我的电脑，进入属性设置，选择高级系统设置进入系统属性，点击环境变量，进入到环境变量设置，编辑系统变量下的 Path 变量，增加以下地址变量：

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\bin\intel64\Debug
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\bin\intel64\Release
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\3rdparty\tbb\bin
```

该路径为默认安装路径，如果不更改安装地址直接使用上方路径即可，如果进行了修改，请将< C:\Program Files (x86)\Intel\>替换为更改的安装路径。

2.2.3 C++项目配置 OpenVINO™

OpenVINO™ 安装完成后，并不能直接在 C++项目中使用，需要配置相关运行环境，主要需要配置包含目录、库目录以及外部依赖项这三方面。

（1）包含目录设置

右击 C++项目，进入到 C++项目的属性设置，如图 2- 4 所示，选择需要配置的项目配置属性（Debug or Release）和平台属性（x64 or Win32），根据自己平台运行要求进行选择。

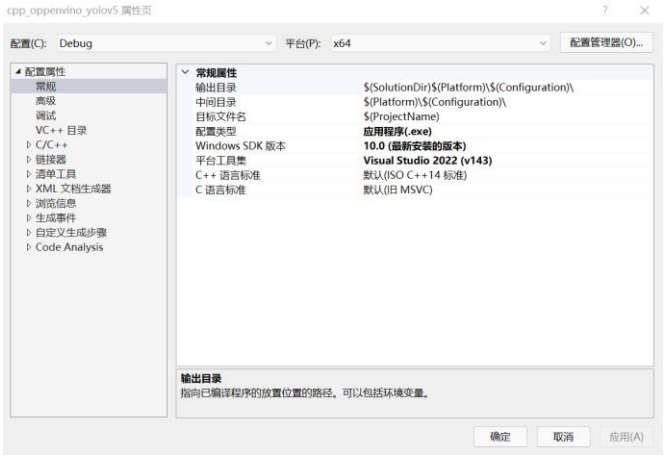


图 2- 4 C++项目属性页

点击 VC++目录，选择包含目录，点击下拉选择，点击编辑，进入到包含目录设置页面，如图 2- 5 所示。

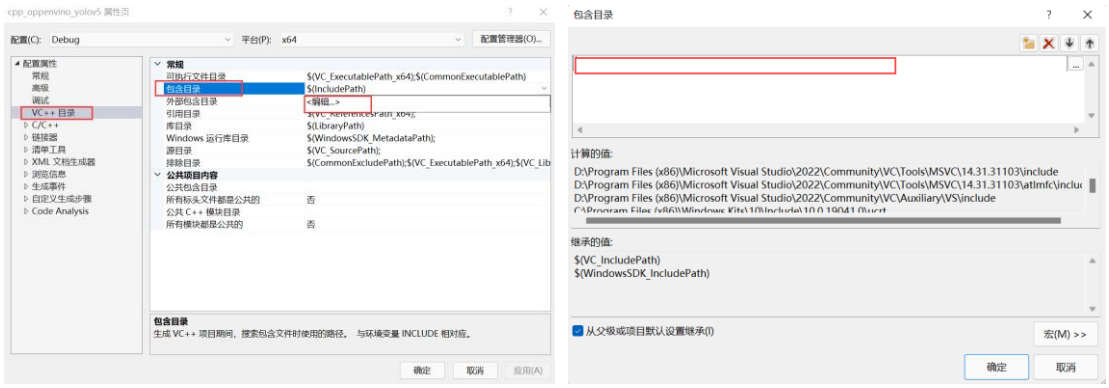


图 2- 5 包含目录设置

在包含目录中添加下面路径：

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\include
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\include\ie
```

< C:\Program Files (x86)\Intel\openvino_2022.1.0.643\>为 OpenVINO™ 安装路径，具体可以根据自己电脑安装的 OpenVINO™ 位置修改。

（2）库目录设置

点击 VC++目录，选择库目录，点击下拉选择，点击编辑，进入到库目录设置页面，如图 2- 6 所示。

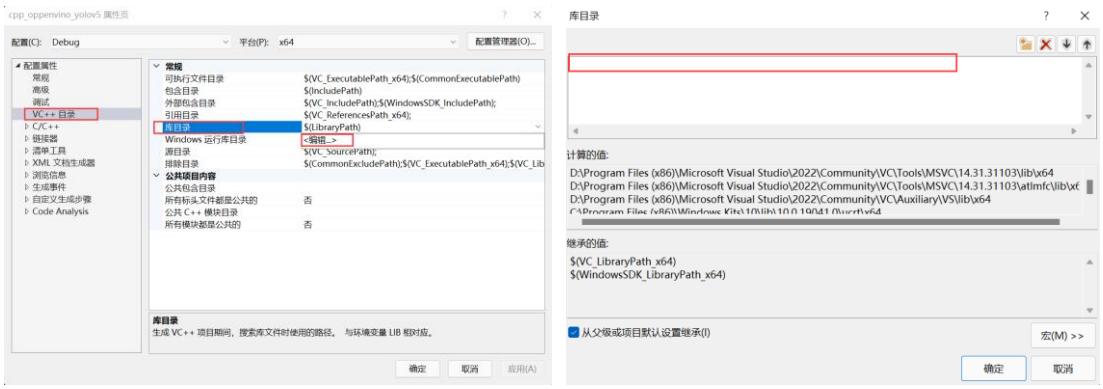


图 2- 6 库目录设置

Debug 模式下，在库目录中添加下面路径：

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\lib\intel64\Debug
```

Release 模式下，在库目录中添加下面路径：

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\lib\intel64\Release
```

（3）附加依赖项设置

点击 VC++目录，选择链接器，点击输入，下拉选择，点击编辑，进入到附加依赖项设置页面，如图 2- 7 所示。

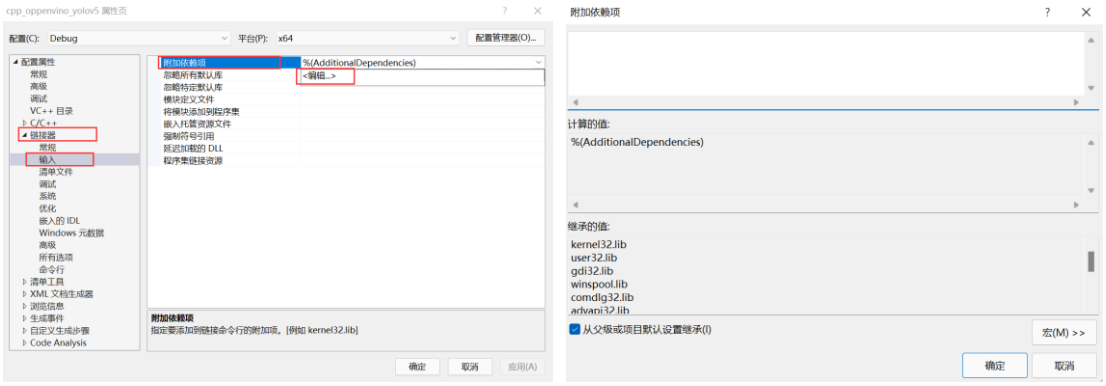


图 2- 7 附加依赖项设置

Debug 模式下，添加该文件名：openvinod.lib；Release 模式下，添加该文件名：openvino.lib。

2.3 NVIDIA TensorRT®™

2.3.1 TensorRT 介绍

NVIDIA TensorRT®™（TensorRT），是由 NVIDIA 推出的 C++语言开发的高性能神经网络深度学习推理的 SDK，包括深度学习推理优化器，并且运行时可为推理应用程序提供低延迟和高吞吐量，其高性能计算能力依赖于 NVIDIA 的图形处理单元。它专注于推理任务，与常用的神经网络学习框架形成互补，可以直接载入这些框架的已训练模型文件，包括 TensorFlow、Caffe、PyTorch、MXNet 等框架，如图 2- 8 所示。

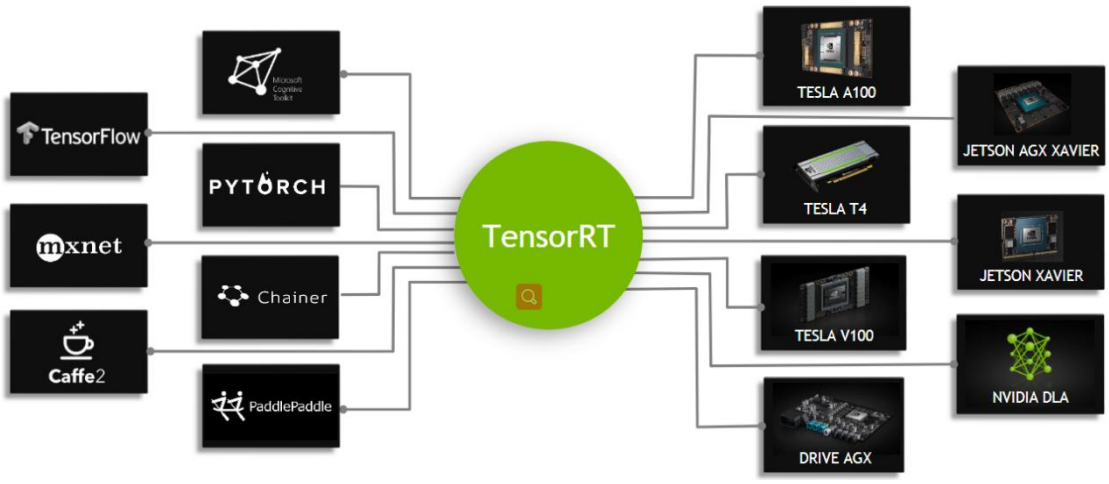


图 2- 8 TensorRT 架构

TensorRT 基于 NVIDIA CUDA 并行编程模型构建，使开发人员能够通过利用 CUDA-X®™ 中的库、开发工具和技术来优化推理用于人工智能、自主机器、高性能计算和图形处理，借助新的 NVIDIA Ampere Architecture GPU，TensorRT 还使用稀疏张量内核来进一步提高性能。

TensorRT 提供 INT8，使用量化感知训练和雨后量化，以及 FP16 优化，用于深度学习推理应用程序的生产部署，如视频流、语音识别、推荐、欺诈检测、文本生成和自然语言处理。降低精确度推断可显著降低应用程序延迟，这是许多实时服务以及自主和嵌入式应用程序的要求。基于 NVIDIA TensorRT 的应用程序在推理过程中的执行速度比仅使用 CPU 的平台快 36 倍，使开发人员能够优化在所有主要框架上训练的神经网络模型，以高精度校准以降低精度，并部署到超大规模数据中心、嵌入式平台或汽车产品平台。

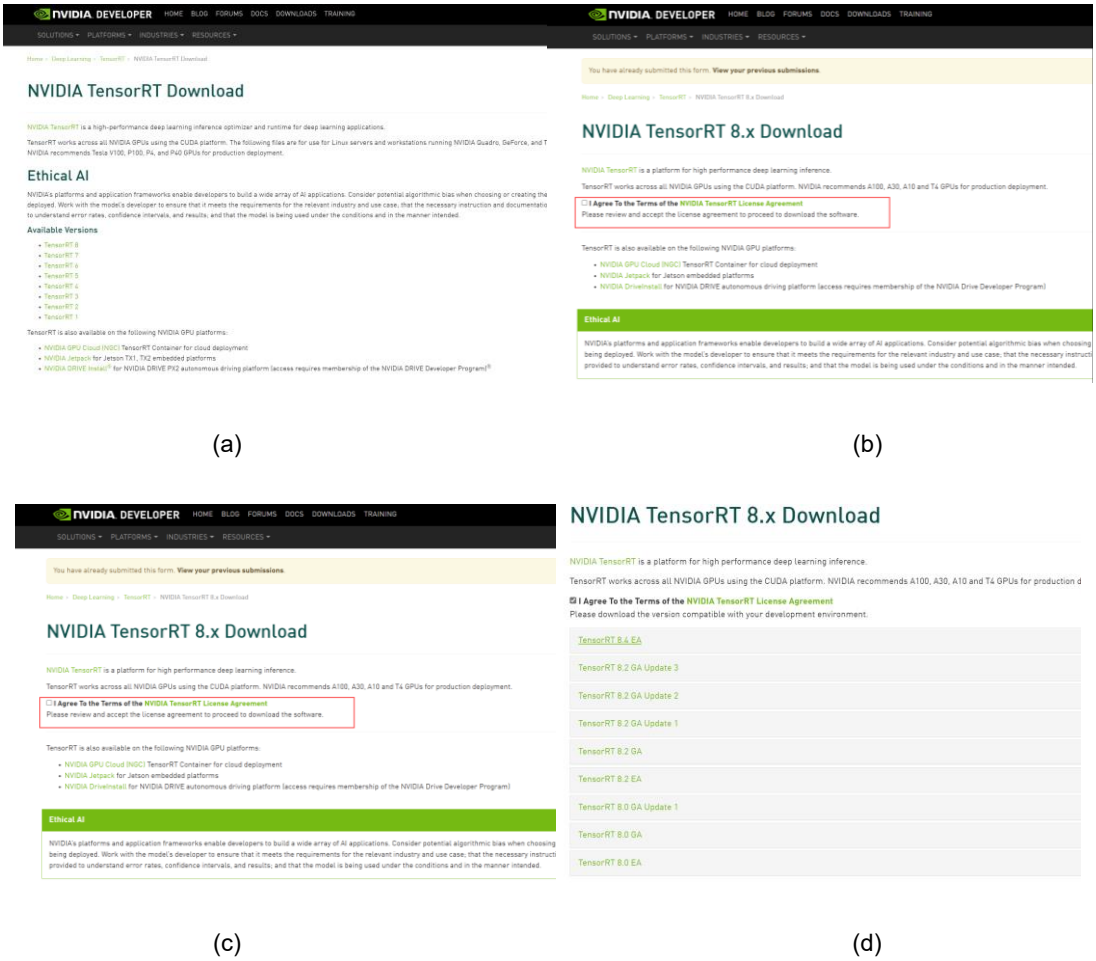
TensorRT 具有以下特性：

- 精度降低：通过量化模型同时保持准确性，使用 FP16 或 INT8 最大限度地提高吞吐量
- 层和张量融合：通过融合内核中的节点来优化 GPU 内存和带宽的使用
- 内核自动调谐：基于目标 GPU 平台选择最佳数据层和算法
- 动态张量记忆：最大限度地减少内存占用，并有效地将内存重用于张量
- 多流执行：使用可扩展的设计并行处理多个输入流
- 时间融合：使用动态生成的内核，在时间步骤中优化递归神经网络

2.3.2 TensorRT 安装

（1）TensorRT 下载

TensorRT 是由英伟达推出的，可通过访问英伟达官网找到，此处我们直接访问下载链接< https://developer.nvidia.com/nvidia-tensorrt-download >，该网站需要注册，并加入 NVIDIA 开发者计划，具体按照网站要求选择，最终跳转到图所示界面。



(a) (b)

(c) (d)

图 2- 9 TensorRT 下载页面

目前 TensorRT 已经更新到版本 8，点击 TensorRT 8，进入到下载页面，点击下载页面<I Agree To the Terms of the NVIDIA TensorRT License Agreement>，如图所示。

点击后会出现图所示界面，我们选择<TensorRT 8.4 EA>，点击后会出现下载文件详细信息选择页面，如图所示

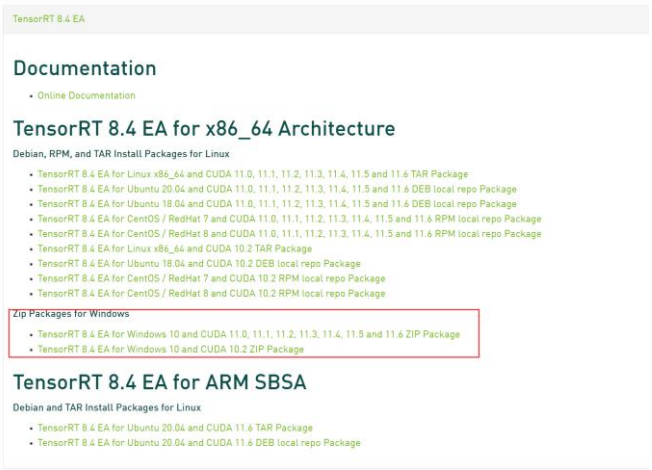


图 2- 10 TensorRT 8.4 EA 版本选择

我们平台为 Windows 平台，并且此处选择也要查看安装包所支持的 CUDA 版本与电脑所安装的是否一致，此处必须一致才可以。查看版本信息可以打开命令提示符窗口，输入以下指令:

```
nvcc --version
```

如图 2- 11 所示，本机电脑 CUDA 版本为 V11.6.55，目前 TensorRT 8.4 EA 支持的最高版本为 11.6，此处便可以直接下载使用。

图 2- 11 CUDA 版本查看方式

如果你的 CUDA 版本过低，当前版本中没有支持，此处需要降低 TensorRT 版本去查找所支持的型号。

文件下载本地后，可以看到下载文件名为：TensorRT-8.4.0.6.Windows10.x86_64.cuda-11.6.cudnn8.3.zip。其中包含信息为：

TensorRT 版本：8.4.0.6；

适用平台：Windows10.x86_64，此处 Windows11 也可使用；

CUDA 版本：11.6；

CUDNN 版本：8.3。

（2）安装软件

文件下好后，只需要将文件解压到本地即可，具体安装位置自行确定，此处我将其解压到< D:\ProgramFiles\TensorRT-8.4.0.6\>文件夹下，后续配置相关信息都与该文江路经有

关。

（3）配置环境变量

在电脑系统环境变量 PATH 变量中增加以下路径，具体操作参考 OpenVINO™ 配置环境变量。

```
D:\ProgramFiles\TensorRT-8.4.0.6\lib
```

2.3.3 C++项目配置 TensorRT

具体操作过程可以参考 OpenVINO™ 设置 C++项目。

（1）包含目录设置

在 C++项目包含目录中添加下述路径：

```
CUDA：C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\include
TensorRT：D:\ProgramFiles\TensorRT-8.4.0.6\include
```

（2）库目录设置

在 C++项目库目录中添加下述路径：

```
CUDA:
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\lib
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\lib\x64
TensorRT:
D:\ProgramFiles\TensorRT-8.4.0.6\lib
```

对于 CUDA 路径，请根据自己电脑安装的位置选择。

（3）附加依赖项设置

在 C++项目附加依赖项中添加下述文件：

```
nvinfer.lib
nvinfer_plugin.lib
nvonnxparser.lib
nvparsers.lib
cublas.lib
cublasLt.lib
cuda.lib
cudadevrt.lib
cudart.lib
cudart_static.lib
cudnn.lib
cudnn64_8.lib
cudnn_adv_infer.lib
cudnn_adv_infer64_8.lib
cudnn_adv_train.lib
cudnn_adv_train64_8.lib
```

```
cudnn_cnn_infer.lib
cudnn_cnn_infer64_8.lib
cudnn_cnn_train.lib
cudnn_cnn_train64_8.lib
cudnn_ops_infer.lib
cudnn_ops_infer64_8.lib
cudnn_ops_train.lib
cudnn_ops_train64_8.lib
cufft.lib
cufftw.lib
curand.lib
cusolver.lib
cusolverMg.lib
cusparse.lib
nppc.lib
nppial.lib
```

```
nppicc.lib
nppidei.lib
nppif.lib
nppig.lib
nppim.lib
nppist.lib
nppisu.lib
nppitc.lib
npps.lib
nvblas.lib
nvjpeg.lib
nvml.lib
nvrtc.lib
OpenCL.lib
```

2.4 OpenCV

2.4.1 OpenCV 简介

OpenCV 是 Open Source Computer Vision Library 的缩写，是一个基于 Apache2.0 许可（开源）发行的跨平台计算机视觉和机器学习软件库，可以运行在 Linux、Windows、Android 和 Mac OS 操作系统上。它轻量级而且高效——由一系列 C 函数和少量 C++ 类构成，同时提供了 Python、Ruby、MATLAB 等语言的接口，实现了图像处理和计算机视觉方面的很多通用算法。

2.4.2 OpenCV 安装

（1）下载并安装 OpenCV

访问 OpenCV 官网 <https://opencv.org/> ，选择 Library 下的 Releases，进入到下载页面，或直接访问<https://opencv.org/releases/> 进入下载页面。

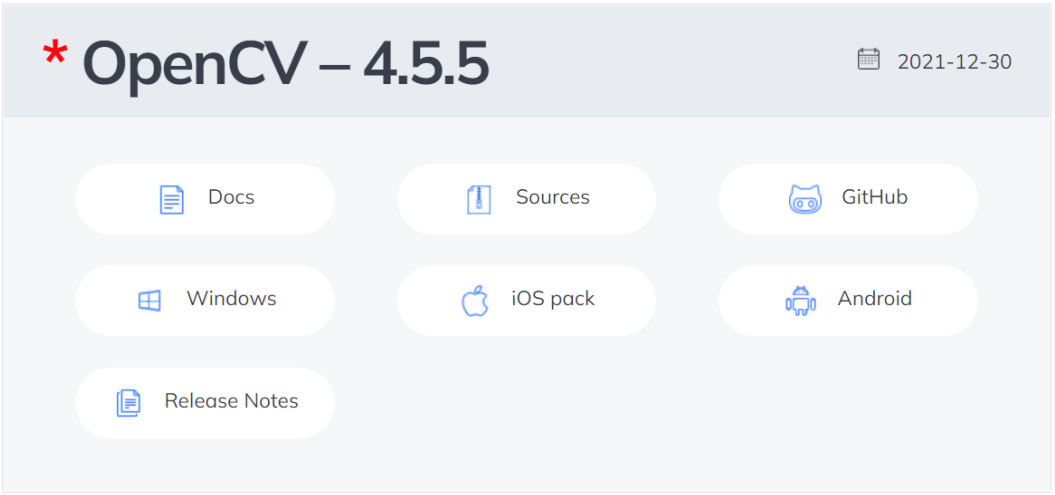


图 2- 12 OpenCV-4.5.5 版本页面

根据负载使用情况，选择 Windows 版本，如所示，跳转页面后，下载文件名为：opencv-4.5.5-vc14_vc15.exe。下载完成后，直接双击打开安装文件，安装完成后，打开安装文件夹，该文件夹下 build、sources 文件夹以及 LICENSE 相关文件，我们所使用的文件在 build 文件夹中。

（2）配置 Path 环境变量

右击我的电脑，进入属性设置，选择高级系统设置进入系统属性，点击环境变量，进入到环境变量设置，编辑系统变量下的 Path 变量，增加以下地址变量：

```
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\bin
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib
E:\OpenCV Source\opencv-4.5.5\build\include
E:\OpenCV Source\opencv-4.5.5\build\include\opencv2
```

其中<E:\OpenCV Source\opencv-4.5.5>为本机安装 OpenCV 安装路径。

2.4.3 C++项目配置 OpenCV

具体操作过程可以参考 OpenVINO™ 设置 C++项目。

（1）包含目录设置

在 C++项目包含目录中添加下述路径：

```
E:\OpenCV Source\opencv-4.5.5\build\include
```

（2）库目录设置

在 C++项目库目录中添加下述路径：

```
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib
```

（3）附加依赖项设置

在 C++项目附加依赖项中添加下述文件：

```
Debug 模式：opencv_world455d.lib
Release 模式：opencv_world455.lib
```

第3章 测试模型

3.1 Yolov5 模型

3.1.1 模型获取与转换

新建一个文件夹，使用命令提示符窗口切换到该页面，输入下面代码，克隆 GitHub 库，并安装相关的依赖项：

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
pip install -r requirements.txt
```

安装完依赖项后，可以通过以下命令进行模型的获取以及转换：

```
python export.py --weights yolov5s.pt --include onnx
```

weights 指的是 yolov5 权重文件，该文件可以下载官方训练好的文件，如果本地没有改文件，会自动在官方下载该文件；include 指的是模型转换的格式，在此处我们将模型转为 onnx 格式。运行后如所示

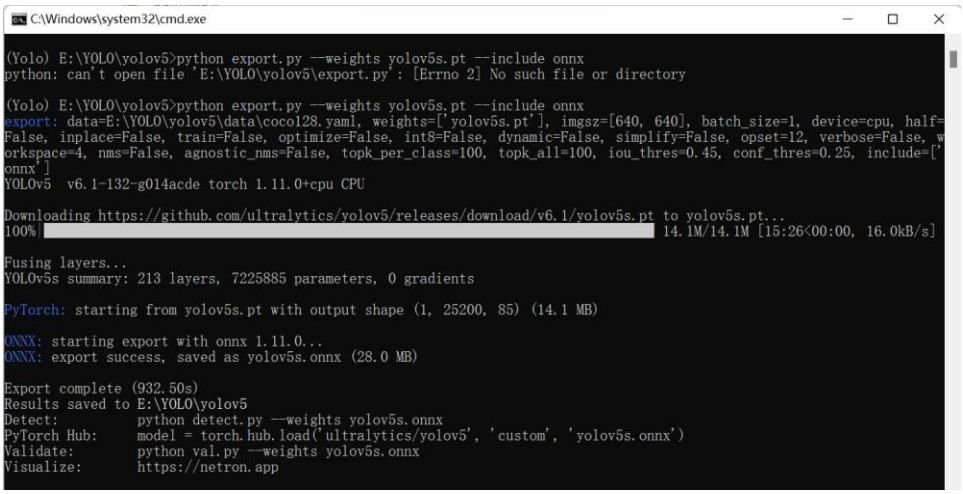


图 3- 1 Yolov5 模型下载与转换

3.1.2 模型相关信息

YOLOv5 模型是 Ultralytics 公司于 2020 年 6 月 9 日公开发布的。YOLOv5 模型是基于 YOLOv3 模型基础上改进而来的，有 YOLOv5s、YOLOv5m、YOLOv5l、YOLOv5x 四个模型。YOLOv5 模型由骨干网络、颈部和头部组成。官网提供的 Yolov5 模型是一个可以识别 80 中物品的训练好的网络，该网络输入为 3×640×640 的图片数据，输出为 25200×85 的识别结果数据，具体信息可以查看表 3- 1。

表 3- 1 Yolov5 模型输入与输出节点信息

节点类型	节点名	节点形状	数据类型	备注
Input	image	[1, 3, 640, 640]	float32	输入网络的图像
Output	output	[1,25200,85]	float32	各个识别结果概率

识别对象	'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'

对与输入数据，需要将输入数据归一化处理，将每个通道数据比 255 即可。对于输出数据，其为一个 25200×85 的数组，其中每 85 个数据为一组，输出一个预测框信息。第 1 个数据为置信值，代表这一组数据的正确度；第 2-5 个数据代表预测框的位置信息，后面绘制矩形框时需要使用到位置信息；第 6-85 个数据，代表对 80 个类别的判断的置信值。

3.2 ResNet50 模型

ResNet 已经被广泛运用于各种特征提取应用中，当深度学习网络层数越深时，理论上表达能力会更强，但是 CNN 网络达到一定的深度后，再加深，分类性能不会提高，而是会导致网络收敛更缓慢，准确率也随着降低，即使把数据集增大，解决过拟合的问题，分类性能和准确度也不会提高。为了解决这一问题，提出了残差学习网络，有效的解决了网络层次增加但网络精度不提升的问题。

飞桨图像识别套件 PaddleClas 是飞桨为工业界和学术界提供的的一个图像识别任务的工具集，该模型经过数据集训练，可以识别多种物品。在该项目中，我们使用 flower 数据集，使用 ResNet50 网络训练识别 102 种花卉，关于该模型的输入与输出节点信息如表 3- 2 所示。

表 3- 2 ResNet50 模型输入与输出节点信息

节点类型	节点名	节点形状	数据类型	备注
Input	x	[1, 3, 224, 224]	float32	输入网络的图像
Output	softmax_1.tmp_0	[1, 102]	float32	各个识别结果概率

关于该模型的训练与导出，可以参考飞桨官方提供的花卉分类模型训练教程（PaddleClas/quick_start_classification_new_user.md at release/2.4 · PaddlePaddle/PaddleClas (github.com)），此处不再详细实现。

第4章 OpenVINO™ 部署 AI 模型实现

4.1 OpenVINO™ 部署模型 C++实现

4.1.1 OpenVINO™ 部署模型基本步骤

经典的一个 OpenVINO™ 部署模型步骤为：读取本地模型、将模型加载到设备、创建推理请求、配置输入数据、模型推理以及处理推理结果，对于新版 OpenVINO™，增加了对 pdmodel 模型以及 onnx 模型的直接读取的支持，并且 pdmodel 模型以及 onnx 模型可以是输入 bath_size 不指定，但 OpenVINO™ 模型推理需要指定所有维度尺寸，因此新版 OpenVINO™ 需要增加设置输入尺寸这一步。

（1）读取本地模型

ov 为新版 OpenVINO™ 的命名空间，新版将所有代码整合到了该命名空间下，在读取本地模型前，需要初始化 Core 对象，这个类代表一个 OpenVINO™ 运行时核心实体；然后调用 read_model()读取本地模型到内存。

```
ov::Core core;
std::shared_ptr<ov::Model> model_ptr = core.read_model(std::string model_path);
```

（2）将模型加载到设备

OpenVINO™ 支持英特尔公司生产推出的各种 CPU 以及 GPU 等设备，因此在加载到设备时，需要根据自己设备进行选择，CPU 设备可以不为英特尔 CPU，不过显卡设备必须为英特尔生产的，否者不能部署。新版可以使用 AUTO 不指定设备，而是在运行时，会根据当前设备选择合适设备进行部署。

```
ov::CompiledModel compiled_model = core.compile_model(std::string model_ptr, "CPU");
```

（3）创建推理请求

这一步主要是创建模型推理请求，后续进行数据配置以及模型推理，都在推理请求上进行，并且模型推理球球可以创建多个，同步进行不同数据的推理。

```
ov::InferRequest infer_request = compiled_model.create_infer_request();
```

（4）设置输入节点尺寸

set_shape()可以对动态输入节点进行固定，固定为指定形状大小，如果该节点形状已经指定，也会按照最终设置的大小确定。

```
ov::Tensor input_image_tensor = infer_request.get_tensor(std::string input_node_name);
input_image_tensor.set_shape(ov::Shape& shape);
```

（5）配置推理输入数据

新版 OpenVINO™ 配置输入数据是，可以直接通过访问节点内存的方式进行填充数据，该方式支持各种数据填充，其最终填充数据为数组数据。因此如果是填充的图片数据，需要在填充前对图片数据进行处理。

```
float* input_tensor_data = input_tensor.data<float>();
for (int i = 0; i < data_size; i++) {
    input_tensor_data[i] = input_data[i];
}
```

（6）处理推理结果

首先是要将输出结果从输出节点内存上对取出来，然后再处理数据；对于不同的模

型，其最后的输出结果是不同的，有不同的处理方式，具体要参考模型的输出要求。

```
const ov::Tensor& output_tensor = infer_request.get_tensor(output_node_name);
float* result_array = output_tensor.data<float>();
```

4.1.2 OpenVINO™ 部署 Yolov5 模型

（1）新建 C++项目

右击解决方案，选择添加新建项目，添加一个 C++空项目，将 C++项目命名为：cpp_oppenvino_yolov5。进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的文件的添加。

右击当前项目，进入属性设置，配置 OpenVINO™ 以及 OpenCV 的属性。

（2）定义 yolov5 模型相关信息

```
std::string model_path = "E:/Text_Model/yolov5/yolov5s.onnx";
std::string image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
std::string lable_path = "E:/Git_space/Al 模型部署开发方式/model/yolov5/lable.txt";
std::string input_node_name = "images";
std::string output_node_name = "output";
```

（3）初始化相关对象

此处主要是初始化 Core 对象、读取本地模型、将模型加载到设备以及创建推理请求。

```
ov::Core core; // core 对象
std::shared_ptr<ov::Model> model_ptr = core.read_model(model_path);
ov::CompiledModel compiled_model = core.compile_model(model_ptr, "CPU");
ov::InferRequest infer_request = compiled_model.create_infer_request();
```

（4）配置输入数据

首先获取输入节点的形状信息，直接使用 get_shape()方法便可以获取输入节点各个维度的信息。

```
ov::Tensor input_image_tensor = infer_request.get_tensor(input_node_name);
int input_H = input_image_tensor.get_shape()[2]; //获得"image"节点的 Height
int input_W = input_image_tensor.get_shape()[3]; //获得"image"节点的 Width
```

接下来就是预处理图片数据，Yolov5 模型输入图片为方型、大小为（640X640）、RGB 通道以及归一化处理。首先将图片放置到方型背景下：

```
cv::Mat image = cv::imread(image_path);; //  读取输入图片
//  将输入图片放置在正方形背景上
int max_side_length = std::max(image.cols, image.rows);
cv::Mat max_image = cv::Mat::zeros(cv::Size(max_side_length, max_side_length), CV_8UC3);
cv::Rect roi(0, 0, image.cols, image.rows);
image.copyTo(max_image(roi));
```

接下来就是转换 RGB 通道以及按照指定大小缩放图片：

```
//  交换 RGB 通道
cv::Mat rgb_image;
cv::cvtColor(max_image, rgb_image, cv::COLOR_BGRA2RGB);
//  缩放至指定大小
```

```
cv::Mat normal_image;
cv::resize(rgb_image, normal_image, cv::Size(input_H, input_W), 0, 0, cv::INTER_LINEAR);
```

然后将图片进行归一化处理：

```
// 将图像归一化
std::vector<cv::Mat> rgb_channels(3);
cv::split(normal_image, rgb_channels);// 分离数据通道
for (auto i = 0; i < rgb_channels.size(); i++) {
    rgb_channels[i].convertTo(rgb_channels[i], CV_32FC1, 1.0 / 255, 0);
}
cv::merge(rgb_channels, normal_image);
```

最后就是将处理完的图片填充到输入节点上：

```
fill_tensor_data_image(input_image_tensor, normal_image);
```

fill_tensor_data_image()方法为自己整架的方法，其主要实现原理是将图片数据展开为一维数组，并逐点将数据写入到内存中。

（5）模型推理

```
infer_request.infer();
```

（6）结果处理

首先是读取推理结果，推理结果在输出节点 Tensor 上的内存中：

```
const ov::Tensor& output_tensor = infer_request.get_tensor(output_node_name);
float* result_array = output_tensor.data<float>();
```

接下来就是处理数据，Yolov5 输出结果为 85x25200 大小的数组，其中没 85 个数据为一组，在该项目中我们提供了专门用于处理 yolov5 数据结果的结果处理类，因此在此处我们只需要调用该结果类即可：

```
ResultYolov5 result;
result.read_class_names(lable_path);
esult.factor = max_side_length / (float)input_H;
cv::Mat result_image = result.yolov5_result(image, result_array);
```

最终输出结果如所示。

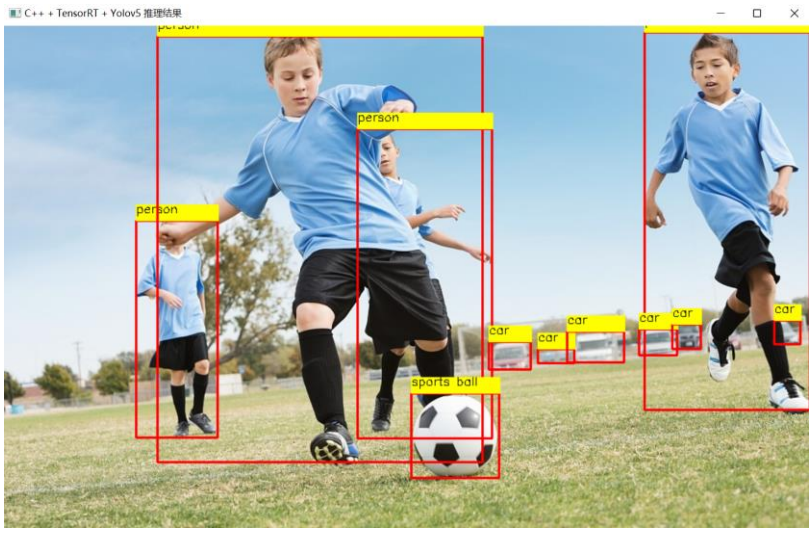


图 4- 1 模型推理结果

4.2 OpenVinoSharp

4.2.1 新建 OpenVINO™ 接口实现文件

右击解决方案，选择添加新建项目，添加一个 C++空项目，将 C++项目命名为：cpp_openvino_api。进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的文件的添加。具体操作如图 4- 2 所示。

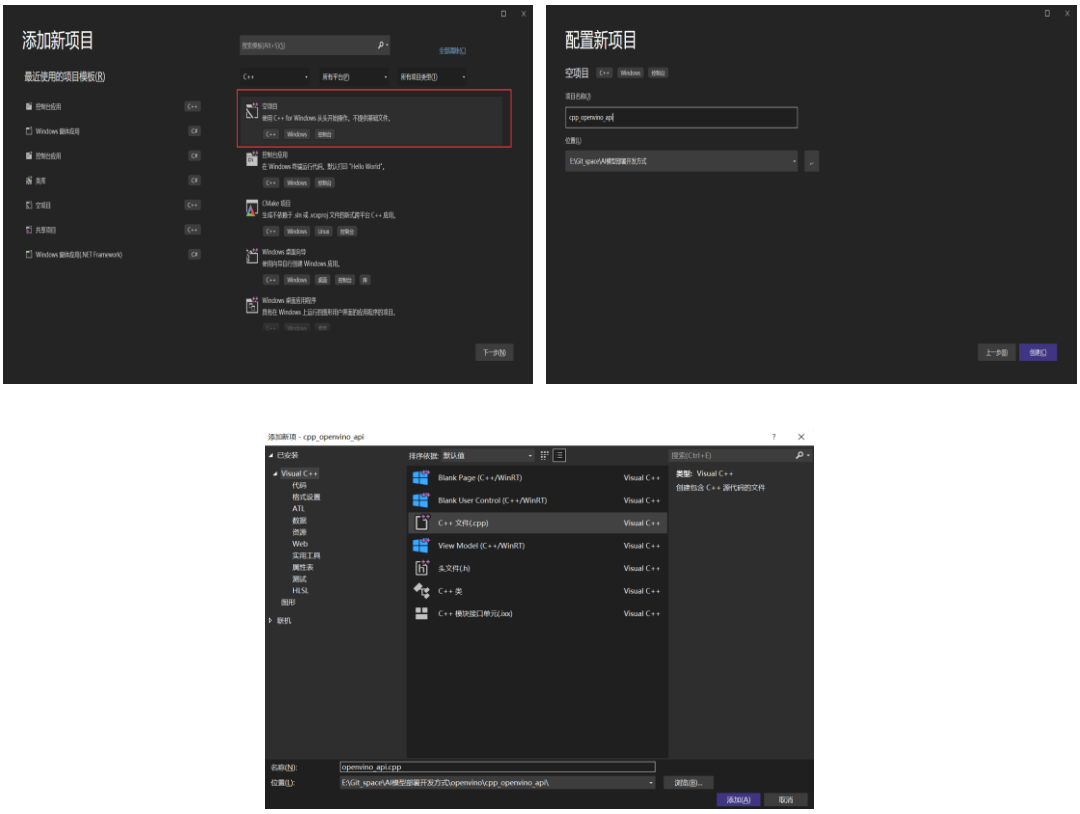


图 4- 2 新建 C++项目

4.2.2 配置 C++项目属性

右击项目，点击属性，进入到属性设置，此处需要设置项目的配置类型包含目录、库目录以及附加依赖项，本次项目选择 Release 模式下运行，因此以 Release 情况进行配置。

（1）设置配置与平台

进入属性设置后，在最上面，将配置改为 Release，平台改为 x64。具体操作如图 4- 3 所示。

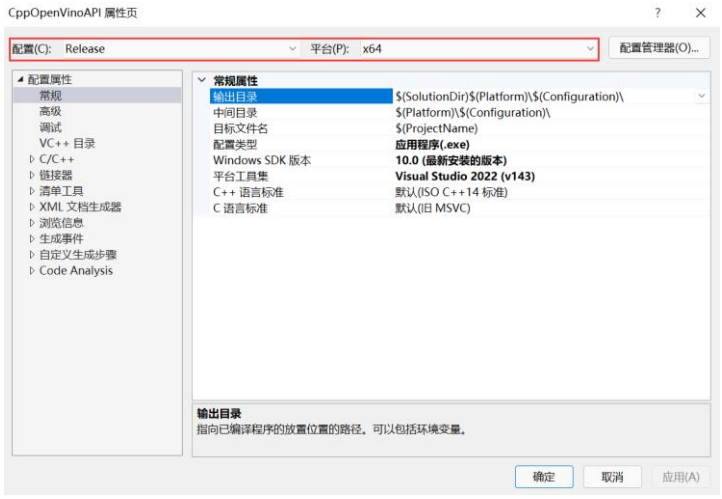


图 4- 3 C++项目属性配置与平台设置

（2）设置常规属性

常规设置下，点击输出目录，将输出位置设置为< $(SolutionDir)dll_import/openvino >，即将生成文件放置在项目文件夹下的 dll 文件夹下；其次将目标文件名修改为：openvinosharp；最后将配置类型改为：动态库(.dll)，让其生成 dll 文件。具体操作如图 4- 4 所示。

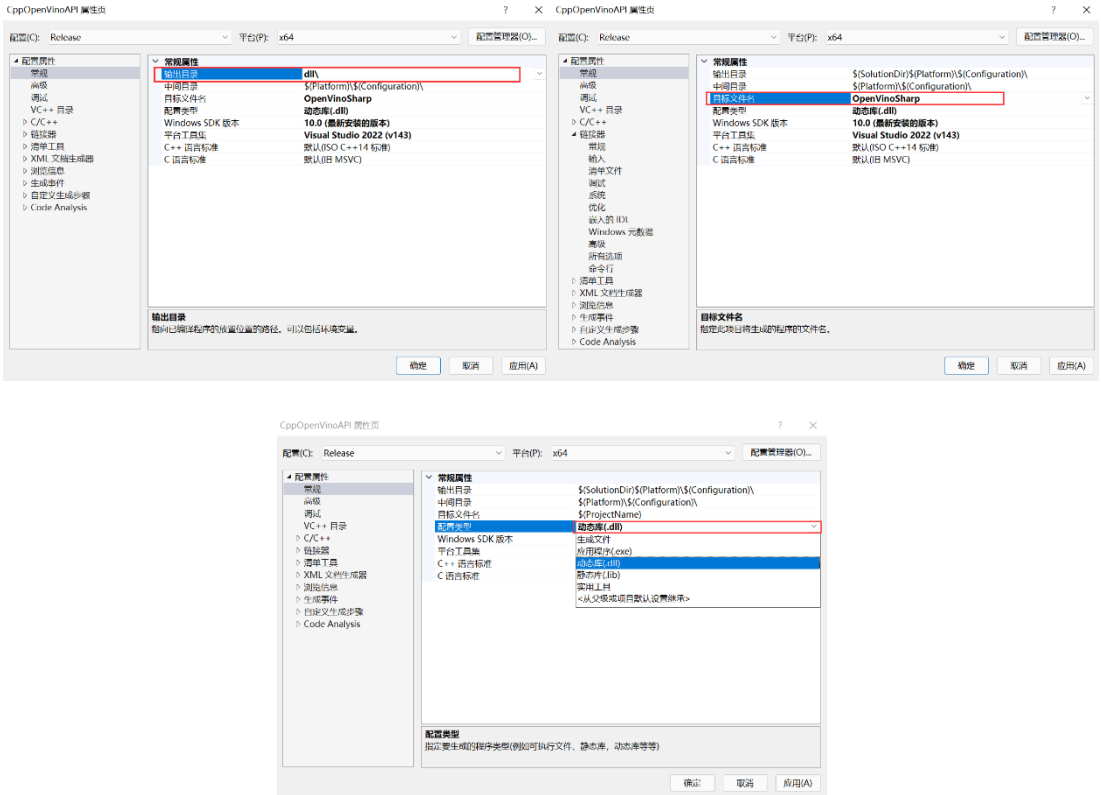


图 4- 4 C++项目常规属性设置

（3）配置附加包

此处需要使用 OpenVINO™ 与 OpenCV 两个外部依赖包，因此需要配置相关设置，具体操作方式按照第二章 OpenVINO™ 配置 C++项目与 OpenCV 配置 C++项目部分。

4.2.3 编写 C++代码

（1）推理引擎结构体

Core 是 OpenVINO™ 工具套件里的推理核心类，该类下包含多个方法，可用于创建推理中所使用的其他类。在此处，需要在各个方法中传递的仅仅是所使用的几个变量，因此选择构建一个推理引擎结构体，用于存放各个变量。

```
// @brief 推理核心结构体
typedef struct openvino_core {
    ov::Core core; // core 对象
    std::shared_ptr<ov::Model> model_ptr; // 读取模型指针
    ov::CompiledModel compiled_model; // 模型加载到设备对象
    ov::InferRequest infer_request; // 推理请求对象
} CoreStruct;
```

其中 Core 是 OpenVINOTM 工具套件里的推理机核心，该模块只需要初始化；shared_ptr<ov::Model>是读取本地模型的方法，新版更新后，该方法发生了较大改动，可支持读取 Paddlepaddle 飞桨模型、onnx 模型以及 IR 模型；CompiledModel 指的是一个已编译的模型类，其主要是将读取的本地模型应用多个优化转换，然后映射到计算内核，由所指定的设备编译模型；InferRequest 是一个推理请求类，在推理中主要用于对推理过程的操作。

（2）接口方法规划

经典的 OpenVINO™ 进行模型推理，一般需要八个步骤，主要是：初始化 Core 对象、读取本地推理模型、配置模型输入&输出、载入模型到执行硬件、创建推理请求、准备输入数据、执行推理计算以及处理推理计算结果。我们根据原有的八个步骤，对步骤进行重新整合，并根据推理步骤，调整方法接口。

对于方法接口，主要设置为：推理初始化、配置输入数据形状、配置输入数据、模型推理、读取推理结果数据以及删除内存地址六个大类，其中配置输入数据形状要细分为配置图片数据形状以及普通数据形状，配置输入数据要细分为配置图片输入数据与配置普通数据输入，读取推理结果数据细分为读取 float 数据和 int 数据，因此，总共有 6 类方法接口，9 个方法接口。

（3）初始化推理模型

OpenVINO™ 推理引擎结构体是联系各个方法的桥梁，后续所有操作都是在推理引擎结构体中的变量上操作的，为了实现数据在各个方法之间的传输，因此在创建推理引擎结构体时，采用的是创建结构体指针，并将创建的结构体地址作为函数返回值返回。推理初始化接口主要整合了原有推理的初始化 Core 对象、读取本地推理模型、载入模型到执行硬件和创建推理请求步骤，并将这些步骤所创建的变量放在推理引擎结构体中。

初始化推理模型接口方法为：

```
extern "C" __declspec(dllexport) void* __stdcall core_init(const wchar_t* model_file_wchar, const wchar_t*
device_name_wchar);
```

该方法返回值为 CoreStruct 结构体指针，其中 model_file_wchar 为推理模型本地地址字符串指针，device_name_wchar 为模型运行设备名指针，在后面使用上述变量时，需要将其转换为 string 字符串，利用 wchar_to_string()方法可以实现将其转换为字符串格式：

```
std::string model_file_path = wchar_to_string(model_file_wchar);
std::string device_name = wchar_to_string(device_name_wchar);
```

模型初始化功能主要包括：初始化推理引擎结构体和对结构体里面定义的其他变量进行赋值操作，其主要是利用 InferEngineStruct 中创建的 Core 类中的方法，对各个变量进行初始化操作：

```
CoreStruct* p = new CoreStruct(); // 创建推理引擎指针
p->model_ptr = p->core.read_model(model_file_path); // 读取推理模型
p->compiled_model = p->core.compile_model(p->model_ptr, "CPU"); // 将模型加载到设备
p->infer_request = p->compiled_model.create_infer_request(); // 创建推理请求
```

（4）配置输入数据形状

在新版 OpenVINO™ 2022.1 中，新增加了对 Paddlepaddle 模型以及 onnx 模型的支持，Paddlepaddle 模型不支持指定指定默认 bath 通道数量，因此需要在模型使用时指定其输入；其次，对于 onnx 模型，也可以在转化时不指定固定形状，因此在配置输入数据前，需要配置输入节点数据形状。其方法接口为：

```
extern "C"  __declspec(dllexport) void* __stdcall set_input_image_sharp(void* core_ptr, const wchar_t*
input_node_name_wchar, size_t * input_size);
extern "C"  __declspec(dllexport) void* __stdcall set_input_data_sharp(void* core_ptr, const wchar_t*
input_node_name_wchar, size_t * input_size);
```

由于需要配置图片数据输入形状与普通数据的输入形状，在此处设置了两个接口，分别设置两种不同输入的形状。该方法返回值是 CoreStruct 结构体指针，但该指针所对应的数据中已经包含了对输入形状的设置。第一个输入参数 core_ptr 是 CoreStruct 指针，在当前方法中，我们要读取该指针，并将其转换为 CoreStruct 类型：

```
CoreStruct* p = (CoreStruct*)core_ptr;
```

input_node_name_wchar 为待设置网络节点名，input_size 为形状数据数组，对图片数据，需要设置 [batch, dim, height, width] 四个维度大小，所以 input_size 数组传入 4 个数据，其设置在形状主要使用 Tensor 类下的 set_shape()方法：

```
std::string input_node_name = wchar_to_string(input_node_name_wchar); // 将节点名转为 string 类型
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name); // 读取指定节点 Tensor
input_image_tensor.set_shape({ input_size[0],input_size[1],input_size[2],input_size[3] }); // 设置节点数据形状
```

（5）配置输入数据

在新版 OpenVINO™ 中，Tensor 类的 T* data()方法，其返回值为当前节点 Tensor 的数据内存地址，通过填充 Tensor 的数据内存，实现推理数据的输入。对于图片数据，其最终也是将其转为一维数据进行输入，不过为方便使用，此处提供了配置图片数据和普通数据的接口，对于输入为图片的方法接口：

```
extern "C"  __declspec(dllexport) void* __stdcall load_image_input_data(void* core_ptr,
    const wchar_t* input_node_name_wchar, uchar * image_data, size_t image_size, int BN_means);
```

该方法返回值是 CoreStruct 结构体指针，但该指针所对应的数据中已经包含了加载的图片数据。第一个输入参数 core_ptr 是 CoreStruct 指针，在当前方法中，我们要读取该指针，并将其转换为 CoreStruct 类型；第二个输入参数 input_node_name_wchar 为待填充节点名，先将其转为 string 字符串：

```
std::string input_node_name = wchar_to_string(input_node_name_wchar);
```

在该项目中，我们主要使用的是以图片作为模型输入的推理网络，模型主要的输入为图片的输入。其图片数据主要存储在矩阵 image_data 和矩阵长度 image_size 两个变量中。需要对图片数据进行整合处理，利用创建的 data_to_mat () 方法，将图片数据读取到 OpenCV

中：

```
cv::Mat input_image = data_to_mat(image_data, image_size);
```

接下来就是配置网络图片数据输入，对于节点输入是图片数据的网络节点，其配置网络输入主要分为以下几步：

首先，获取网络输入图片大小。

使用 InferRequest 类中的 get_tensor ()方法，获取指定网络节点的 Tensor，其节点要求输入大小在 Shape 容器中，通过获取该容器，得到图片的长宽信息：

```
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name);
int input_H = input_image_tensor.get_shape()[2]; //获得"image"节点的 Height
int input_W = input_image_tensor.get_shape()[3]; //获得"image"节点的 Width
```

其次，按照输入要求，处理输入图片。

在这一步，我们除了要按照输入大小对图片进行放缩之外，还要根据 PaddlePaddle 对模型输入的要求进行处理。因此处理图片其主要分为交换 RGB 通道、放缩图片以及对图片进行归一化处理。在此处我们借助 OpenCV 来实现。

OpenCV 读取图片数据并将其放在 Mat 类中，其读取的图片数据是 BGR 通道格式，PaddlePaddle 要求输入格式为 RGB 通道格式，其通道转换主要靠一下方式实现：

```
cv::cvtColor(input_image, blob_image, cv::COLOR_BGR2RGB);
```

接下来就是根据网络输入要求，对图片进行压缩处理：

```
cv::resize(blob_image, blob_image, cv::Size(input_H, input_W), 0, 0, cv::INTER_LINEAR);
```

最后就是对图片进行归一化处理，其主要处理步骤就是减去图像数值均值，并除以方差。BN_means 为数据归一化处理方式选择，主要提供了 PaddlePaddle 数据处理方式以及普通数据处理方式，其中 BN_means = 1 时为 PaddlePaddle 数据处理方式，当 BN_means = 2 时，为普通数据处理方式，即将数据整除 255。查询 PaddlePaddle 模型对图片的处理，其均值 mean = [0.485, 0.456, 0.406]，方差 std = [0.229, 0.224, 0.225]，利用 OpenCV 中现有函数，对数据进行归一化处理：

```
std::vector<cv::Mat> rgb_channels(3);
cv::split(blob_image, rgb_channels); //  分离图片数据通道
if (BN_means == 0) {
        std::vector<float> mean_values{ 0.485 * 255, 0.456 * 255, 0.406 * 255 };
        std::vector<float> std_values{ 0.229 * 255, 0.224 * 255, 0.225 * 255 };
        for (auto i = 0; i < rgb_channels.size(); i++) {
            rgb_channels[i].convertTo(rgb_channels[i], CV_32FC1, 1.0 / std_values[i], (0.0 - mean_values[i]) /
std_values[i]);
            }
      }
else if (BN_means == 1){
        for (auto i = 0; i < rgb_channels.size(); i++) {
            rgb_channels[i].convertTo(rgb_channels[i], CV_32FC1, 1.0 / 255, 0);
            }
      }
```

最后，将图片数据输入到模型中。

在此处，我们重写了网络赋值方法，并将其封装到 fill_tensor_data_image(ov::Tensor& input_tensor, const cv::Mat& input_image)方法中，input_tensor 为模型输入节点 Tensor 类，input_image 为处理过的图片 Mat 数据。因此节点赋值只需要调用该方法即可：

```
fill_tensor_data_image(input_image_tensor, blob_image);
```

对于普通数据的输入，其方法接口如下：

```
extern "C"    __declspec(dllexport) void* __stdcall load_input_data(void* core_ptr, const wchar_t*
input_node_name_wchar, float* input_data);
```

与配置图片数据不同点，在于输入数据只需要输入 input_data 数组即可。其数据处理哦在外部实现，只需要将处理后的数据填充到输入节点的数据内存中即可，通过调用自定义的 fill_tensor_data_float(ov::Tensor& input_tensor, float* input_data, int data_size) 方法即可实现：

```
std::string input_node_name = wchar_to_string(input_node_name_wchar);
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name); //  读取指定节点 tensor
int input_size = input_image_tensor.get_shape()[1]; //获得输入节点的长度
fill_tensor_data_float(input_image_tensor,input_data, input_size); //  将数据填充到 tensor 数据内存上
```

（6）模型推理

上一步中我们将推理内容的数据输入到了网络中，在这一步中，我们需要进行数据推理，这一步中我们留有一个推理接口：

```
extern "C"    __declspec(dllexport) void* __stdcall core_infer(void* core_ptr)
```

进行模型推理，只需要调用 CoreStruct 结构体中的 infer_request 对象中的 infer()方法即可：

```
CoreStruct* p = (CoreStruct*)core_ptr;
p->infer_request.infer();
```

（7）读取推理数据

上一步我们对数据进行了推理，这一步就需要查询上一步推理的结果。对于我们所使用的模型输出，主要有 float 数据和 int 数据，对此，留有了两种数据的查询接口，其方法为：

```
extern "C"    __declspec(dllexport) void __stdcall read_infer_result_F32(void* core_ptr, const wchar_t*
output_node_name_wchar, int data_size, float* infer_result);
extern "C"    __declspec(dllexport) void __stdcall read_infer_result_I32(void* core_ptr, const wchar_t*
output_node_name_wchar, int data_size, int* infer_result);
```

其中 data_size 为读取数据长度，infer_result 为输出数组指针。读取推理结果数据与加载推理数据方式相似，依旧是读取输出节点处数据内存的地址：

```
const ov::Tensor& output_tensor = p->infer_request.get_tensor(output_node_name);
const float* results = output_tensor.data<const float>();
```

针对读取整形数据，其方法一样，只是在转换类型时，需要将其转换为整形数据即可。我们读取的初始数据为二进制数据，因此要根据指定类型转换，否则数据会出现错误。将数据读取出来后，将其放在数据结果指针中，并将所有结果赋值到输出数组中：

```
for (int i = 0; i < data_size; i++) {
    *inference_result = results[i];
    inference_result++;
}
```

（8）删除推理核心结构体指针

推理完成后，我们需要将在内存中创建的推理核心结构地址删除，防止造成内存泄露，影响电脑性能，其接口该方法为：

```
extern "C"    __declspec(dllexport) void __stdcall core_delet(void* core_ptr);
```

在该方法中，我们只需要调用 delete 命令，将结构体指针删除即可。

4.2.4 编写模块定义文件

我们在定义接口方法时，在原有方法的基础上，增加了 extern "C" 、__declspec(dllexport) 以及__stdcall 三个标识，其主要原因是为了让编译器识别我们的输出方法。其中，extern "C" 是指示编译器这部分代码按 C 语言（而不是 C++）的方式进行编译；__declspec(dllexport)用于声明导出函数、类、对象等供外面调用；__stdcall 是一种函数调用约定。通过上面三个标识，我们在 C++种所写的接口方法，会在 dll 文件中暴露出来，并且可以实现在 C#中的调用。

不过上面所说内容，我们在编辑器中可以通过模块定义文件(.def)所实现，在模块定义文件中，添加以下代码：

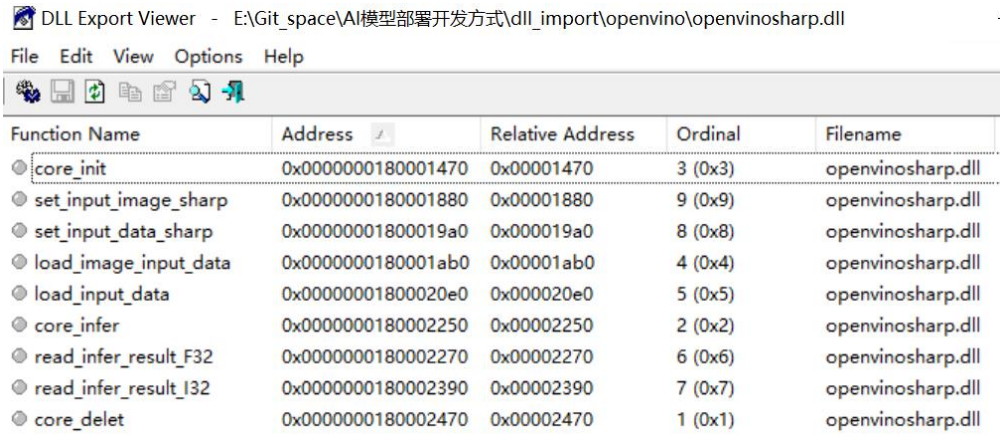
```
LIBRARY
        "openvinosharp"
EXPORTS
        core_init
        set_input_image_sharp
        set_input_data_sharp
        load_image_input_data
        load_input_data
        core_infer
        read_infer_result_F32
        read_infer_result_I32
        core_delet
```

LIBRARY 后所跟的为输出文件名，EXPORTS 后所跟的为输出方法名。仅需要以上语句便可以替代 extern "C" 、 __declspec(dllexport) 以及__stdcall 的使用。

4.2.5 生成 dll 文件

前面我们将项目配置输出设置为了生成 dll 文件，因此该项目不是可以执行的 exe 文件，只能生成不能运行。右键项目，选择重新生成/生成。在没有错误的情况下，会看到项目成功的提示。可以看到 dll 文件在解决方案同级目录下.\dll_import\openvino\文件夹下。

使用 dll 文件查看器打开 dll 文件，如所示；可以看到，我们创建的四个方法接口已经暴露在 dll 文件中。



Function Name	Address	Relative Address	Ordinal	Filename
core_init	0x0000000180001470	0x00001470	3 (0x3)	openvinosharp.dll
set_input_image_sharp	0x0000000180001880	0x00001880	9 (0x9)	openvinosharp.dll
set_input_data_sharp	0x00000001800019a0	0x000019a0	8 (0x8)	openvinosharp.dll
load_image_input_data	0x0000000180001ab0	0x00001ab0	4 (0x4)	openvinosharp.dll
load_input_data	0x00000001800020e0	0x000020e0	5 (0x5)	openvinosharp.dll
core_infer	0x0000000180002250	0x00002250	2 (0x2)	openvinosharp.dll
read_infer_result_F32	0x0000000180002270	0x00002270	6 (0x6)	openvinosharp.dll
read_infer_result_I32	0x0000000180002390	0x00002390	7 (0x7)	openvinosharp.dll
core_delet	0x0000000180002470	0x00002470	1 (0x1)	openvinosharp.dll

openvino.dll 文件方法输出目录

4.3 C#构建 Core 类

4.3.1 新建 C#类库

右击解决方案，添加->新建项目，选择添加 C#类库，项目名命名为 csharp_openvino_class，项目框架根据电脑中的框架选择，此处使用的是.NET 5.0。新建完成后，然后右击项目，选择添加->新建项，选择类文件，添加 Core.cs 和 NativeMethods.cs 两个类文件。

4.3.2 引入 dll 文件中的方法

在 NativeMethods.cs 文件下，我们通过[DllImport()]方法，将 dll 文件中所有的方法读取到 C#中。读取方式如下：

```
[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr core_init(string model_file, string device_name);
```

其中 openvino_dll_path 为 dll 文件路径，CharSet = CharSet.Unicode 代表支持中文编码格式字符串，CallingConvention = CallingConvention.Cdecl 指示入口点的调用约定为调用方清理堆栈。

上述所列出的为初始化推理模型，dlii 文件接口在匹配时，是通过方法名字匹配的，因此，方法名要保证与 dll 文件中一致。其次就是方法的参数类型要进行对应，在上述方法中，函数的返回值在 C++中为 void* ，在 C#中对应的为 IntPtr 类型，输入参数中，在 C++中为 wchar_t* 字符指针，在 C#中对应的为 string 字符串。通过方法名与参数类型一一对应，在 C#可以实现对方法的调用。其他方法的引用类似，在此处不在一一赘述，具体可以参照项目提供的源代码。

4.3.3 创建 Core 类

为了更方便地调用我们通过 dll 引入的 OpenVINO™ 方法，减少使用时的函数方法接口，我们在 C#中重新组建我们自己的推理类，命名为 Class Core，其主要成员变量和方法如图 4- 5 所示。

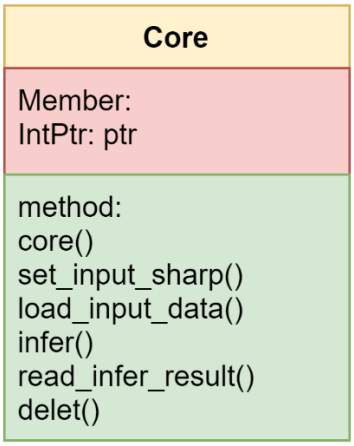


图 4- 5 Core 类图

（1）成员变量

在 Core.类中，我们只需要创建一个地址变量，作为 Core 类的成员变量，用于接收接口函数返回的推理核心指针，该成员变量我们只需要在当前类下访问，因此将其设置为私有变量：

```
private IntPtr ptr = new IntPtr();
```

（2）构造函数

在类的初始化时，我们需要输入模型地址以及设备类型，通过掉用 dll 文件中引入的方法，获取初始化指针，对成员变量进行赋值，实现类的初始化：

```
public Core(string model_file, string device_name) {
    ptr = NativeMethods.core_init(model_file, device_name);
}
```

（3）节点形状设置方法

在构建设置数据输入形状时，我们需要提供的为节点名以及形状数据，为了简化该方法，我们合并了图片形状设置与普通数据形状设置接口，通过判断输入数组的长度，来确定是对那一个形状的设置。

```
public void set_input_sharp(string input_node_name, ulong[] input_size){
// 获取输入数组长度
    int length = input_size.Length;
    if (length == 4) {// 长度为 4，判断为设置图片输入的输入参数，调用设置图片形状方法
        ptr = NativeMethods.set_input_image_sharp(ptr, input_node_name, ref input_size[0]);
    }
    else if (length == 2){// 长度为 2，判断为设置普通数据输入的输入参数，调用设置普通数据形状方法
        ptr = NativeMethods.set_input_data_sharp(ptr, input_node_name, ref input_size[0]);
    }
   else{ // 为防止输入发生异常，直接返回
        return;
    }
}
```

（4）加载推理数据

在这一步，我们除了要加载图片数据，还需要加载普通的数据，因此为了简化接口名，我们使用函数重载的方式，配置加载推理数据方法，主要是通过方法输入数据的不同来区分。

```
public void load_input_data(string input_node_name, float[] input_data) {
    ptr = NativeMethods.load_input_data(ptr, input_node_name, ref input_data[0]);
}
public void load_input_data(string input_node_name, byte[] image_data, ulong image_size, int BN_means){
    ptr = NativeMethods.load_image_input_data(ptr, input_node_name, ref image_data[0], image_size,
BN_means);
}
```

（5）模型推理方法

```
public void infer(){
    ptr = NativeMethods.core_infer(ptr);
}
```

（6）读取结果数据

在读取结果数据这一步，我们提供了整形数据与浮点型数据的支持，因此在此处使用泛型编程的思想，将两种方式写在一起，通过设定的数据类型来指定待读取数据类型，来调用相关的接口。

```
public T[] read_infer_result<T>(string output_node_name, int data_size){
    // 获取设定类型
    string t = typeof(T).ToString();
    // 新建返回值数组
    T[] result = new T[data_size];
    if (t == "System.Int32"){ // 读取数据类型为整形数据
        int[] inference_result = new int[data_size];
        NativeMethods.read_infer_result_I32(ptr, output_node_name, data_size, ref inference_result[0]);
        result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
        return result;
    }
    else{ // 读取数据类型为浮点型数据
        float[] inference_result = new float[data_size];
        NativeMethods.read_infer_result_F32(ptr, output_node_name, data_size, ref inference_result[0]);
        result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
        return result;
        }
    }
```

（7）内存销毁

```
public void delet(){
    NativeMethods.core_delet(ptr);
}
```

4.3.4 编译 Core 类库

右击项目，点击生成/重新生成，出现如下**错误!未找到引用源。**所示，表示编译成功。

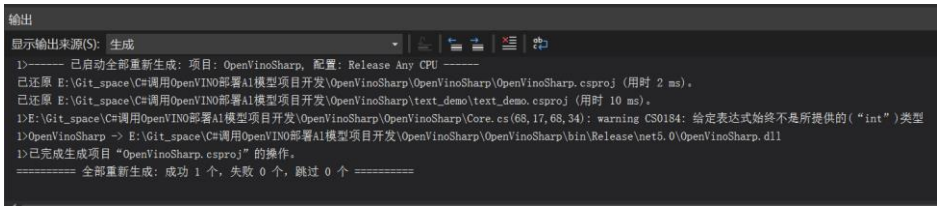


图 4- 6 Core 类编译输出

4.4 OpenVINO™ 部署模型 C#实现

4.4.1 C#项目配置

（1）配置 OpenVinoSharp

新建项目后，右击项目，添加->项目引用，选择我们上一步创建的 csharp_openvino_class 类库项目，点击确定。csharp_process_infer_result 为我们创建的一个模型结果处理类，此处不做过多讲解，如图 4- 7 所示。

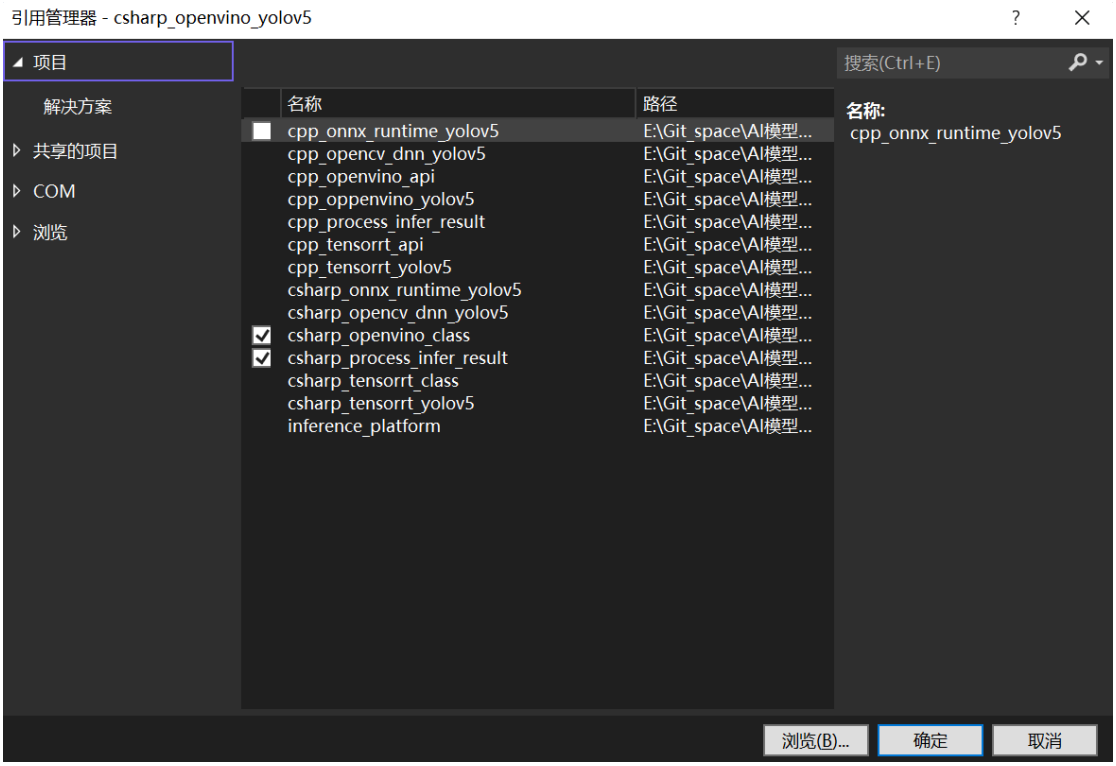


图 4- 7 添加 OpenVinoSharp 项目引用

（2）配置 OpenCvSharp

右击项目，点击管理 NuGet 程序包，在出现的页面，点击浏览，在搜索况中输入 opencvsharp4，点击搜索后，会看到 OpenCvSharp4 以及 OpenCvSharp4.runtime.win 两个程序包，如图 4- 8 所示。

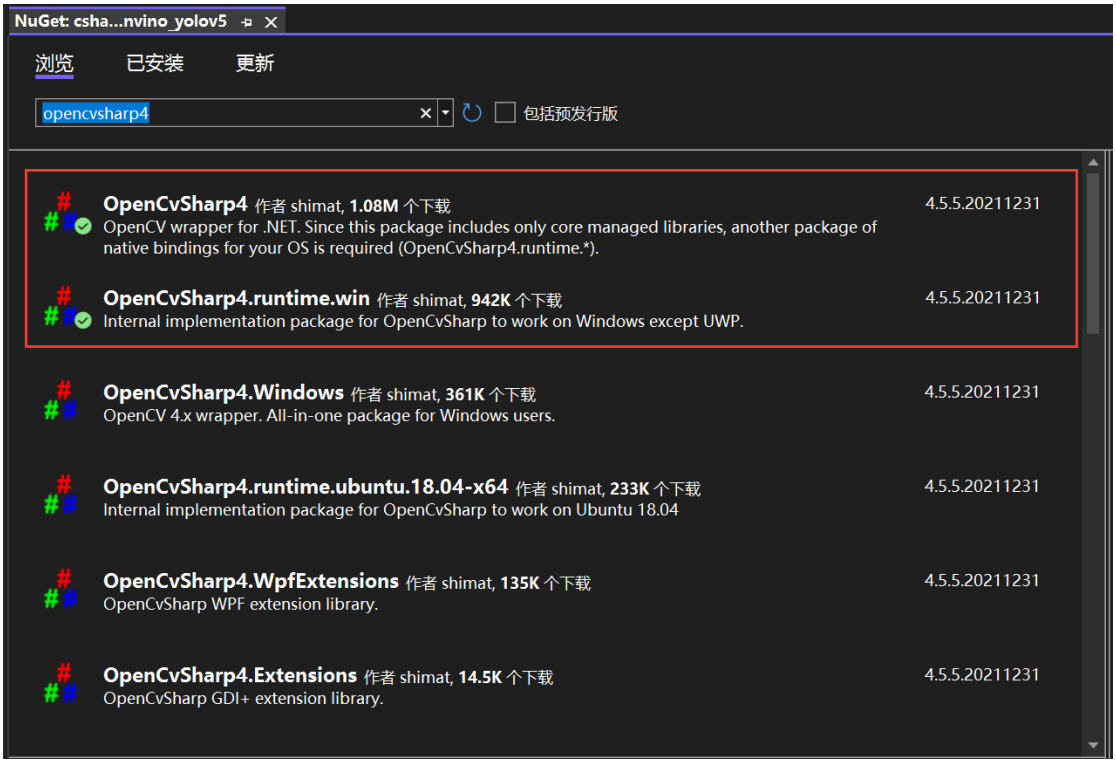


图 4- 8 OpenCvSharp 程序包安装

点击 OpenCvSharp4 安装包后，在右侧会出现如图 4- 9 页面，点击安装，然后默认选择安装即可。OpenCvSharp4.runtime.win 也是默认安装即可。

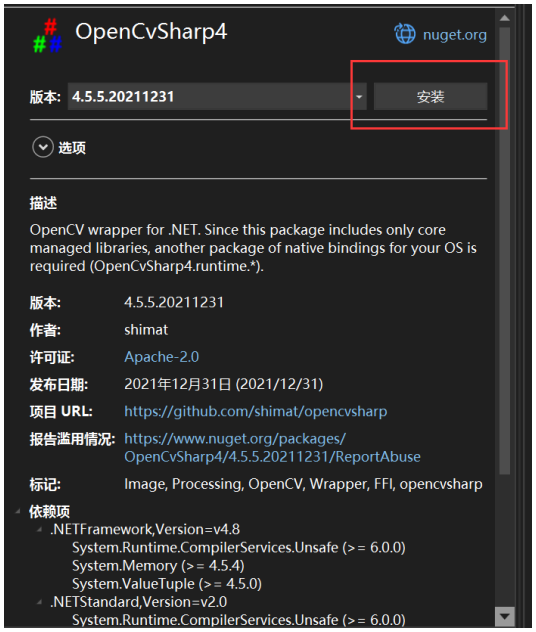


图 4- 9 OpenCvSharp4 安装页面

4.4.2 OpenVINO™ 部署 Yolov5 模型

（1）引入模型相关信息

```
string model_path = "E:/Text_Model/yolov5/yolov5s.onnx";
string image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
string input_node_name = "images";
string output_node_name = "output";
```

（2）初始化推理核心类

```
Core core = new Core(model_path, "CPU");
```

（3）加载推理图片数据

由于 Yolov5 模型已经固定输入数据大小，因此此处不需要在设置大小，根据 Yolov5 模型要求输入，进行图片预处理。

```
Mat image = new Mat(image_path);
// 将图片放在矩形背景下
Mat max_image = Mat.Zeros(new Size(1024, 1024), MatType.CV_8UC3);
Rect roi = new Rect(0, 0, image.Cols, image.Rows);
image.CopyTo(new Mat(max_image, roi));
```

此处通过 OpenCvSharp 将图片数据读取到内存中，并将图片放置在正方形背景下，后续的其他数据处理会在封装的 C++程序中实现，因此此处不需要再做进一步处理。接下来将图片转为矩阵数据，方便数据能在 C++与 C#之间传输。

```
byte[] image_data = new byte[2048 * 2048 * 3];
ulong image_size = new ulong();
image_data = max_image.ImEncode(".bmp");
image_size = Convert.ToUInt64(image_data.Length);
```

接下来直接调用 load_input_data()加载输入数据方法，加载输入数据。

```
core.load_input_data(input_node_name, image_data, image_size, 1);
```

（4）模型推理

```
core.infer();
```

（5）结果处理

首先是读取输出结果，yolov5 模型数据输出为浮点型数据，因此只需要调用 read_infer_result()方法读取数据即可。

```
float[] result_array = core.read_infer_result<float>(output_node_name, 25200 * 85);
```

然后就是处理读取后的数据，在此处我们提供了一个专门针对 Yolov5 结果处理类，用于处理结果，此处我们只需要调用该类，便可以实现。

```
ResultYolov5 result = new ResultYolov5();
result.read_class_names(lable_path);
result.factor = (float)(image.Cols > image.Rows ? image.Cols : image.Rows) / (float)640;
Mat result_image = result.process_resule(image,result_array);
```

最终输出结果为处理完的结果图片，如图 4- 1 所示。

第5章 TensorRT 部署 AI 模型实现

5.1 TensorRT 部署模型 C++实现

5.1.1 TensorRT 部署模型基本步骤

经典的一个 TensorRT 部署模型步骤为：onnx 模型转 engine、读取本地模型、创建推理引擎、创建推理上下文、创建 GPU 显存缓冲区、配置输入数据、模型推理以及处理推理结果。

（1）onnx 模型转 engine

TensorRT 支持多种模型文件，不过随着 onnx 模型的发展，目前多种模型框架都将 onnx 模型当作中间转换格式，是的该模型结构变得越来越通用，因此 TensorRT 目前主要在更新的就是针对该模型的转换。TensorRT 是可以直接读取 engine 文件，对于 onnx 模型需要进行一些列转换配置，转为 engine 引擎才可以进行后续的推理，因此在进行模型推理前，需要先进行模型的转换。项目中已经提供了转换方法接口，此处不作详细赘述。

（2）读取本地模型

此处读取本地模型为读取上一步保存在本地的 engine 二进制文件，将模型文件信息读取到内存中。该文件保存了模型的所有信息以及电脑的配置信息，因此该模型文件不支持在不同电脑上使用。

```
std::ifstream file_ptr(model_path_engine, std::ios::binary);
size_t size = 0;
file_ptr.seekg(0, file_ptr.end);       // 将读指针从文件末尾开始移动 0 个字节
size = file_ptr.tellg();  // 返回读指针的位置，此时读指针的位置就是文件的字节数
file_ptr.seekg(0, file_ptr.beg);       // 将读指针从文件开头开始移动 0 个字节
char* model_stream = new char[size];
file_ptr.read(model_stream, size);
file_ptr.close();
```

（3）创建推理引擎

首先需要初始化日志记录接口类，该类用于创建后续反序列化引擎使用；然后创建反序列化引擎，其主要作用是允许对序列化的功能上不安全的引擎进行反序列化，接下调用反序列化引擎来创建推理引擎，这一步只需要输入上一步中读取的模型文件数据以及长度即可。

```
// 日志记录接口
Logger logger;
// 反序列化引擎
nvinfer1::IRuntime* runtime = nvinfer1::createInferRuntime(logger);
// 推理引擎
nvinfer1::ICudaEngine* engine = runtime->deserializeCudaEngine(model_stream, size);
```

（4）创建推理上下文

这里的推理上下文与 OpenVINO 中的推理请求相似，为后面进行模型推理的类。

```
nvinfer1::IExecutionContext* context = engine->createExecutionContext();
```

（5）创建 GPU 显存缓冲区

TensorRT 是利用英伟达显卡进行模型推理的，但是我们的推理数据以及后续处理数据

是在内存中实现的，因此需要创建显存缓冲区，用于输入推理数据以及读取推理结果数据。

```
// 创建 GPU 显存缓冲区
void** data_buffer = new void* [num_ionode];
// 创建 GPU 显存输入缓冲区
int input_node_index = engine->getBindingIndex(input_node_name);
cudaMalloc(&(data_buffer[input_node_index]), input_data_length * sizeof(float));
// 创建 GPU 显存输出缓冲区
int output_node_index = engine->getBindingIndex(output_node_name);
cudaMalloc(&(data_buffer[output_node_index]), output_data_length * sizeof(float));
```

（6）配置输入数据

配置输入数据时只需要调用 cudaMemcpyAsync()方法，便可将 cuda 流数据加载到与 i 里模型上。但数据需要根据模型要求进行预处理，除此以外需要将数据结果加入到 cuda 流中。

```
// 创建输入 cuda 流
cudaStream_t stream;
cudaStreamCreate(&stream);
std::vector<float> input_data(input_data_length);
memcpy(input_data.data(), BN_image.ptr<float>(), input_data_length * sizeof(float));
// 输入数据由内存到 GPU 显存
cudaMemcpyAsync(data_buffer[input_node_index], input_data.data(), input_data_length * sizeof(float),
cudaMemcpyHostToDevice, stream);
```

（7）模型推理

```
context->enqueueV2(data_buffer, stream, nullptr);
```

（8）处理推理结果

我们最后处理数据是在内存上实现的，首先需要将数据由显存读取到内存中。

```
float* result_array = new float[output_data_length];
cudaMemcpyAsync(result_array, data_buffer[output_node_index], output_data_length * sizeof(float),
cudaMemcpyDeviceToHost, stream);
```

接下来就是根据模型输出结果进行数据处理，不同的模型会有不同的数据处理方式。

5.1.2 TensorRT 部署 Yolov5 模型

（1）新建 C++项目

右击解决方案，选择添加新建项目，添加一个 C++空项目，将 C++项目命名为：cpp_tensorrt_yolov5。进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的文件的添加。

右击当前项目，进入属性设置，配置 TensorRT 以及 OpenCV 的属性。

（2）定义 yolov5 模型相关信息

```
const char* model_path_onnx = "E:/Text_Model/yolov5/yolov5s.onnx";
const char* model_path_engine = "E:/Text_Model/yolov5/yolov5s.engine";
```

```
const char* image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
std::string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
const char* input_node_name = "images";
const char* output_node_name = "output";
int num_ionode = 2;
```

（3）读取本地模型信息

```
std::ifstream file_ptr(model_path_engine, std::ios::binary);
    if (!file_ptr.good()) {
        std::cerr << "文件无法打开，请确定文件是否可用！" << std::endl;
}
size_t size = 0;
file_ptr.seekg(0, file_ptr.end);    // 将读指针从文件末尾开始移动 0 个字节
size = file_ptr.tellg();  // 返回读指针的位置，此时读指针的位置就是文件的字节数
file_ptr.seekg(0, file_ptr.beg);    // 将读指针从文件开头开始移动 0 个字节
char* model_stream = new char[size];
file_ptr.read(model_stream, size);
file_ptr.close();
```

（4）初始化推理引擎

在此处我们需要初始化反序列化引擎以及推理引擎，并创建用于推理的上下文。

```
Logger logger;
// 反序列化引擎
nvinfer1::IRuntime* runtime = nvinfer1::createInferRuntime(logger);
// 推理引擎
nvinfer1::ICudaEngine* engine = runtime->deserializeCudaEngine(model_stream, size);
// 上下文
nvinfer1::IExecutionContext* context = engine->createExecutionContext();
```

（5）创建 GPU 显存缓冲区

GPU 显存缓冲区的数量主要与模型的输入输出节点有关，我们在此处只需要按照模型输入输出的节点数量进行设置。

```
void** data_buffer = new void* [num_ionode];
// 创建 GPU 显存输入缓冲区
int input_node_index = engine->getBindingIndex(input_node_name);
nvinfer1::Dims input_node_dim = engine->getBindingDimensions(input_node_index);
size_t input_data_length = input_node_dim.d[1]* input_node_dim.d[2] * input_node_dim.d[3];
cudaMalloc(&(data_buffer[input_node_index]), input_data_length * sizeof(float));
// 创建 GPU 显存输出缓冲区
int output_node_index = engine->getBindingIndex(output_node_name);
nvinfer1::Dims output_node_dim = engine->getBindingDimensions(output_node_index);
size_t output_data_length = output_node_dim.d[1] * output_node_dim.d[2] ;
cudaMalloc(&(data_buffer[output_node_index]), output_data_length * sizeof(float));
```

（6）配置模型输入

首先对输入图片按照模型数据输入要求进行处理，首先是将图片数据复制到正方形背

景中，然后交换 RGB 通道、缩放至指定大小以及归一化处理，在 OpenCV 中，blobFromImage()方法可以直接实现上述功能。

```
// 图象预处理 - 格式化操作
cv::Mat image = cv::imread(image_path);
int max_side_length = std::max(image.cols, image.rows);
cv::Mat max_image = cv::Mat::zeros(cv::Size(max_side_length, max_side_length), CV_8UC3);
cv::Rect roi(0, 0, image.cols, image.rows);
image.copyTo(max_image(roi));
// 将图像归一化，并放缩到指定大小
cv::Size input_node_shape(input_node_dim.d[2], input_node_dim.d[3]);
cv::Mat BN_image = cv::dnn::blobFromImage(max_image, 1 / 255.0, input_node_shape, cv::Scalar(0, 0, 0), true, false);
```

接下来创建 cuda 流，将处理后的数据放置在 input_data 容器里；最后直接使用 cudaMemcpyAsync()方法，将输入数据输送到显存。

```
// 创建输入 cuda 流
cudaStream_t stream;
cudaStreamCreate(&stream);
std::vector<float> input_data(input_data_length);
memcpy(input_data.data(), BN_image.ptr<float>(), input_data_length * sizeof(float));
// 输入数据由内存到 GPU 显存
cudaMemcpyAsync(data_buffer[input_node_index], input_data.data(), input_data_length * sizeof(float), cudaMemcpyHostToDevice, stream);
```

（7）模型推理

```
context->enqueueV2(data_buffer, stream, nullptr);
```

（8）处理推理结果

首先读取推理结果数据，主要是将 GPU 显存上的推理数据结果赋值到内存上，方便后续对数据的进一步处理。

```
float* result_array = new float[output_data_length];
cudaMemcpyAsync(result_array, data_buffer[output_node_index], output_data_length * sizeof(float), cudaMemcpyDeviceToHost, stream);
```

接下来就是处理数据，Yolov5 输出结果为 85x25200 大小的数组，其中没 85 个数据为一组，在该项目中我们提供了专门用于处理 yolov5 数据结果的结果处理类，因此在此处我们只需要调用该结果类即可：

```
ResultYolov5 result;
result.factor = max_side_length / (float) input_node_dim.d[2];
result.read_class_names(lable_path);
cv::Mat result_image = result.yolov5_result(image, result_array);
```

5.2 TensorRTSharp

5.2.1 新建 TensorRT 接口实现文件

右击解决方案，选择添加新建项目，添加一个 C++空项目，将 C++项目命名为：cpp_tensorrt_api。进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的

文件的添加。具体操作如所示。

5.2.2 配置 C++项目属性

右击项目，点击属性，进入到属性设置，此处需要设置项目的配置类型包含目录、库目录以及附加依赖项，本次项目选择 Release 模式下运行，因此以 Release 情况进行配置。

（1）设置常规属性

进入属性设置后，在最上面，将配置改为 Release，平台改为 x64。

常规设置下，点击输出目录，将输出位置设置为< $(SolutionDir)dll_import/tensorrt >，即将生成文件放置在项目文件夹下的 dll 文件夹下；其次将目标文件名修改为：tensorrtsharp；最后将配置类型改为：动态库(.dll)，让其生成 dll 文件。

（2）配置附加包

此处需要使用 TensorRT 与 OpenCV 两个外部依赖包，因此需要配置相关设置，具体操作方式按照第二章 TensorRT 配置 C++项目与 OpenCV 配置 C++项目部分。

5.2.3 编写 C++代码

（1）推理引擎结构体

Logger 是 TensorRT 工具套件用于创建 IBuilder、IRuntime 或 IRefitter 实例的记录器，该类中 log()方法未进行实例化，因此需要将其实例化，重写该类中的 log()方法。

```
class Logger : public nvinfer1::ILogger{
    void log(Severity severity, const char* message)    noexcept{
        if (severity != Severity::kINFO)
            std::cout << message << std::endl;
    }
} gLogger;
```

为了实现模型推理能在各个接口中传递推理的相关配置信息，所以将相关重要类或结构体整合到 NvinferStruct 结构体中，如下：

```
typedef struct tensorRT_nvinfer {
    Logger logger;
    nvinfer1::IRuntime* runtime;
    nvinfer1::ICudaEngine* engine;
    nvinfer1::IExecutionContext* context;
    cudaStream_t stream;
    void** data_buffer;
} NvinferStruct;
```

Iruntime 为 TensorRT 反序列化引擎，允许对序列化的功能上不安全的引擎进行反序列化，该类中 deserializeCudaEngine()方法可以重构本地保存的模型推理文件；IcudaEngine 为创建的网络上执行推理的引擎，保存了网络模型的相关信息； IexecutionContext 为用于模型推理的上下文，是最终执行推理类；cudaStream_t 为 CUDA stream 标志，主要用于后面在 GPU 显存上传输数据使用；data_buffer 为 GPU 显存输入/输出缓冲内存位置，用于在显存上读取和输入数据。

（2）接口方法规划

TensorRT 进行模型推理，一般需要十个步骤，主要是：初始化 Logger 对象、创建反序列化引擎、读取本地推理模型并初始化推理引擎、创建用于推理上下文、创建 GPU 显存输

入/输出缓冲区、准备输入数据、将输入数据从内存加载到显存、执行推理计算以、从显存读取推理结果到内存和处理推理计算结果。我们根据原有的十个步骤，对步骤进行重新整合，并根据推理步骤，调整方法接口。

对于方法接口，主要设置为：推理引擎初始化、创建 GPU 显存输入/输出缓冲区、加载图片输入数据到缓冲区、模型推理、读取推理结果数据以及删除内存地址六个接口，目前 TensorRT 模型推理接口只允许图片输入，暂不支持其他数据的配置。

（3）ONNX 模型转换

TensorRT 几乎可以支持所有常用的深度学习框架，对于 caffe 和 TensorFlow 来说，TensorRT 可以直接解析他们的网络模型；对于 caffe2，pytorch，mxnet，chainer，CNTK 等框架则是首先要将模型转为 ONNX 的通用深度学习模型，然后对 ONNX 模型做解析。目前 TensorRT 主要是在更新的是 ONNX 模型转换，通过内置 API 将 ONNX 模型转换为 TensorRT 可以直接读取的 engine 文件；engine 文件既包含了模型的相关信息，又包含了转换设备的配置信息，因此转换后的 engine 文件不可以跨设备使用。

模型转换接口方法为：

```
extern "C"    __declspec(dllexport) void __stdcall onnx_to_engine(const wchar_t* onnx_file_path_wchar,
        const wchar_t* engine_file_path_wchar, int type);
```

onnx_file_path_wchar 为 ONNX 格式的本地模型地址，engine_file_path_wchar 为转换后的模型保存地址，type 为模型保存的精度类型，当 type＝1 时保存为 FP16 格式，type＝2 时保存为 INT8 格式。

首先创建构建器，用于创建 config、network、engine 的其他对象的核心类，获取 cuda 内核目录以获取最快的实现。

```
nvinfer1::IBuilder* builder = nvinfer1::createInferBuilder(gLogger);
```

其次就是将 ONNX 模型解析为 TensorRT 网络定义的对象，explicit_batch 为指定与按位或组合的网络属性，network 本地定义的网络结构，该结构支持直接读取 ONNX 网络结构到 TensorRT 格式。

```
const auto explicit_batch = 1U <<
static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
nvinfer1::INetworkDefinition* network = builder->createNetworkV2(explicit_batch);
nvonnxparser::IParser* parser = nvonnxparser::createParser(*network, gLogger);
parser->parseFromFile(onnx_file_path.c_str(), 2);
```

接下来就是创建生成器配置对象，config 主要需要设置工作空间长度以及模型的精度，此处提供 FP16 以及 INT8 格式。

```
nvinfer1::IBuilderConfig* config = builder->createBuilderConfig();
config->setMaxWorkspaceSize(16 * (1 << 20));
if (type == 1) {
        config->setFlag(nvinfer1::BuilderFlag::kFP16);
}
if (type == 2) {
        config->setFlag(nvinfer1::BuilderFlag::kINT8);
}
```

在读取完本地模型和配置完成相关设置后，就可以构建模型推理引擎，通过调用 builder 构建器下的 buildEngineWithConfig()方法实现。

```
nvinfer1::ICudaEngine* engine = builder->buildEngineWithConfig(*network, *config);
```

此处只需要模型转换，因此接下载将推理引擎转为文件流，保存到本地，后续模型推理时只需要直接读取本地保存的推理引擎文件即可。

```
nvinfer1::IHostMemory* model_stream = engine->serialize();
file_ptr.write(reinterpret_cast<const char*>(model_stream->data()), model_stream->size());
```

最后一步就是销毁前面所创建的地址对象，销毁的时候需要按照创建的先后顺序销毁。

```
model_stream->destroy();
engine->destroy();
network->destroy();
parser->destroy();
```

（4）初始化推理模型

TensorRT 推理引擎结构体是联系各个方法的桥梁，后续实现模型信息以及配置相关参数进行传递都是在推理引擎结构上实现的，为了实现数据在各个方法之间的传输，因此在创建推理引擎结构体时，采用的是创建结构体指针，并将创建的结构体地址作为函数返回值返回。推理初始化接口主要整合了原有推理的初始化 NvinferStruct 对象、读取本地推理模型、初始化反序列化引擎、初始化推理引擎、创建上下文以及创建创建 GPU 数据缓冲区，并将这些步骤所创建的变量放在推理引擎结构体中。

初始化推理模型接口方法为：

```
extern "C"    __declspec(dllexport) void* __stdcall nvinfer_init(const wchar_t* engine_filename_wchar, int
num_ionode);
```

该方法返回值为 NvinferStruct 结构体指针，其中 engine_filename_wchar 为推理模型本地地址字符串指针，在后面使用上述变量时，需要将其转换为 string 字符串，利用 wchar_to_string()方法可以实现将其转换为字符串格式：

```
std::string engine_filename = wchar_to_string(engine_filename_wchar);
```

首先第一步通过文件流方式读取本地模型文件，将模型文件读取到内存中：

```
std::ifstream file_ptr(engine_filename, std::ios::binary);
if (!file_ptr.good()) {
        std::cerr << "文件无法打开，请确定文件是否可用！" << std::endl;
}
size_t size = 0;
file_ptr.seekg(0, file_ptr.end);       // 将读指针从文件末尾开始移动 0 个字节
size = file_ptr.tellg();   // 返回读指针的位置，此时读指针的位置就是文件的字节数
file_ptr.seekg(0, file_ptr.beg);       // 将读指针从文件开头开始移动 0 个字节
char* model_stream = new char[size];
file_ptr.read(model_stream, size);
// 关闭文件
file_ptr.close();
```

其次对模型进行初始化，模型初始化功能主要包括：初始化推理引擎结构体和对结构体里面定义的其他变量进行赋值操作，其主要是 NvinferStruct 中各个变量进行初始化操作：

```
NvinferStruct* p = new NvinferStruct(); // 创建推理核心结构体，初始化变量
p->runtime = nvinfer1::createInferRuntime(gLogger); // 初始化反序列化引擎
p->engine = p->runtime->deserializeCudaEngine(model_stream, size); // 初始化推理引擎
```

```
p->context = p->engine->createExecutionContext(); // 创建上下文
p->data_buffer = new void* [num_ionode]; // 创建 gpu 数据缓冲区
```

最后一步就是删除文件流数据，防止出现内存泄漏。

（5）创建 GPU 显存输入/输出缓冲区

TensorRT 主要是使用英伟达显卡 CUDA 在显存上进行模型推理的，因此需要在显存上创建输入输出的缓存区。其创建创建 GPU 显存输入/输出缓冲区方法接口为：

```
extern "C"    __declspec(dllexport) void* __stdcall creat_gpu_buffer(void* nvinfer_ptr,
        const wchar_t* node_name_wchar, size_t data_length);
```

其中 nvinfer_ptr 为 NvinferStruct 结构体指针，为第一步初始化后返回的指针，在该方法中，只需要重新将其转换为 NvinferStruct 类型即可：

```
NvinferStruct* p = (NvinferStruct*)nvinfer_ptr;
```

node_name_wchar 为输入或输出节点名字符串，在此处我们只需要配置输入和输出的节点数据缓存区，并利用 getBindingIndex()方法获取节点的序号，用于指定缓存区的位置：

```
const char* node_name = wchar_to_char(node_name_wchar);
int node_index = p->engine->getBindingIndex(node_name);
```

data_length 为缓存区数据的长度，其数据长度为节点数据大小，然后直接调用 cudaMalloc()方法创建 GPU 显存输入/输出缓冲区

```
cudaMalloc(&(p->data_buffer[node_index]), data_length * sizeof(float));
```

（6）配置图片输入数据

TensorRT 将数据加载到网络输入比较简便，只需要调用 cudaMemcpyAsync()方法便可实现，对于此处，我们只设置了图片数据输入的情况，其实现方法接口为：

```
extern "C"    __declspec(dllexport) void* __stdcall load_image_data(void* nvinfer_ptr,
        const wchar_t* node_name_wchar, uchar * image_data, size_t image_size, int BN_means)；
```

该方法返回值是 NvinferStruct 结构体指针，但该指针所对应的数据中已经包含了加载的图片数据。第一个输入参数 nvinfer_ptr 是 NvinferStruct 指针，在当前方法中，我们要读取该指针，并将其转换为 CoreStruct 类型；第二个输入参数 node_name_wchar 为待填充节点名，先将其转为 char 指针，并查询该节点的序列号：

```
const char* node_name = wchar_to_char(node_name_wchar);
int node_index = p->engine->getBindingIndex(node_name);
```

在该项目中，我们主要使用的是以图片作为模型输入的推理网络，模型主要的输入为图片的输入。其图片数据主要存储在矩阵 image_data 和矩阵长度 image_size 两个变量中。需要对图片数据进行整合处理，利用创建的 data_to_mat () 方法，将图片数据读取到 OpenCV 中：

```
cv::Mat input_image = data_to_mat(image_data, image_size);
```

接下来就是配置网络图片数据输入，对于节点输入是图片数据的网络节点，其配置网络输入主要分为以下几步：

首先，获取网络输入图片大小。

使用 engine 中的 getBindingDimensions ()方法，获取指定网络节点的维度信息，其节点要求输入大小在 node_dim 容器中，通过读取得到图片的长宽信息：

```
nvinfer1::Dims node_dim = p->engine->getBindingDimensions(node_index);
```

```
int node_shape_w = node_dim.d[2];
int node_shape_h = node_dim.d[3];
cv::Size node_shape(node_shape_w, node_shape_h);
size_t node_data_length = node_shape_w * node_shape_h;
```

其次，按照输入要求，处理输入图片。

在这一步，我们除了要按照输入大小对图片进行放缩之外，还要对输入数据进行归一化处理。因此处理图片其主要分为交换 RGB 通道、放缩图片以及对图片进行归一化处理。在此处我们借助 OpenCV 来实现。

对与数据归一化处理方式，我们在此处提供了两种处理方式，一种是百度飞桨归一化处理方式，另一种为普通数据处理方式，主要通过 BN_means 指定实现。对于普通数据处理方式，方式比较简单，OpenCV 中有专门的方法可以实现，该方法可以直接实现交换 RGB 通道、放缩图片以及对图片进行归一化处理，我们通过调用该方式：

```
BN_image = cv::dnn::blobFromImage(input_image, 1 / 255.0, node_shape, cv::Scalar(0, 0, 0), true, false);
```

另一种为百度飞桨归一化处理处理方式，第一部需要交换 RGB 通道

```
cv::cvtColor(input_image, input_image, cv::COLOR_BGR2RGB);
```

接下来就是根据网络输入要求，对图片进行压缩处理：

```
cv::resize(input_image, BN_image, node_shape, 0, 0, cv::INTER_LINEAR);
```

最后就是对图片进行归一化处理，其主要处理步骤就是减去图像数值均值，并除以方差。查询 PaddlePaddle 模型对图片的处理，其均值 mean = [0.485, 0.456, 0.406]，方差 std = [0.229, 0.224, 0.225]，利用 OpenCV 中现有函数，对数据进行归一化处理：

```
std::vector<cv::Mat> rgb_channels(3);
cv::split(BN_image, rgb_channels); //  分离图片数据通道
std::vector<float> mean_values{ 0.485 * 255, 0.456 * 255, 0.406 * 255 };
std::vector<float> std_values{ 0.229 * 255, 0.224 * 255, 0.225 * 255 };
for (auto i = 0; i < rgb_channels.size(); i++) {
    rgb_channels[i].convertTo(rgb_channels[i], CV_32FC1, 1.0 / std_values[i], (0.0 - mean_values[i]) /
std_values[i]);
}
cv::merge(rgb_channels, BN_image);
```

最后，将图片数据输入到模型中。

TensorRT 将输入数据加载到显存中需要通过 cuda 流方式，首先船舰一个异步流，并将输入数据赋值到输入流中：

```
cudaStreamCreate(&p->stream);
std::vector<float> input_data(node_data_length * 3);
memcpy(input_data.data(), BN_image.ptr<float>(), node_data_length * 3 * sizeof(float));
```

然后直接调用 cudaMemcpyAsync()方法，将输入数据加载到显存上：

```
cudaMemcpyAsync(p->data_buffer[node_index], input_data.data(), node_data_length * 3 * sizeof(float),
cudaMemcpyHostToDevice, p->stream);
```

（7）模型推理

上一步中我们将推理内容的数据输入到了网络中，在这一步中，我们需要进行数据推理，

实现模型推理的方法接口为：

```
extern "C"    __declspec(dllexport) void* __stdcall infer(void* nvinfer_ptr)
```

进行模型推理，只需要调用 NvinferStruct 结构体中的 context 对象中的 enqueueV2 ()方法即可：

```
NvinferStruct* p = (NvinferStruct*)nvinfer_ptr;
p->context->enqueueV2(p->data_buffer, p->stream, nullptr);
```

（8）读取推理数据

上一步我们对数据进行了推理，这一步就需要查询上一步推理的结果。对于我们所使用的模型输出，主要有 float 数据，其方法为：

```
extern "C"    __declspec(dllexport) void __stdcall read_infer_result(void* nvinfer_ptr,
        const wchar_t* node_name_wchar, float* output_result, size_t node_data_length)；
```

其中 node_data_length 为读取数据长度，output_result 为输出数组指针。读取推理结果数据与加载推理数据方式相似，主要是将显存上数据赋值到内存上。首先需要获取输入节点的索引值：

```
const char* node_name = wchar_to_char(node_name_wchar);
int node_index = p->engine->getBindingIndex(node_name);
```

接下来在本地创建内存放置结果数据，然后调用 cudaMemcpyAsync()方法，将显存上数据赋值到内存上：

```
std::vector<float> output_data(node_data_length * 3);
cudaMemcpyAsync(output_data.data(), p->data_buffer[node_index], node_data_length * sizeof(float),
cudaMemcpyDeviceToHost, p->stream);
```

将数据读取出来后，将其放在数据结果指针中，并将所有结果赋值到输出数组中：

```
for (int i = 0; i < node_data_length; i++) {
        *output_result = output_data[i];
        output_result++;
}
```

（8）删除推理核心结构体指针

推理完成后，我们需要将在内存中创建的推理核心结构地址删除，防止造成内存泄露，影响电脑性能，其接口该方法为：

```
extern "C"    __declspec(dllexport) void __stdcall nvinfer_delete(void* nvinfer_ptr);
```

在该方法中，我们只需要调用 delete 命令，将结构体指针删除即可。

5.2.4 编写模块定义文件

在模块定义文件中，添加以下代码：

```
LIBRARY
        "tensorrtsharp"
EXPORTS
        onnx_to_engine
        nvinfer_init
        creat_gpu_buffer
        load_image_data
```

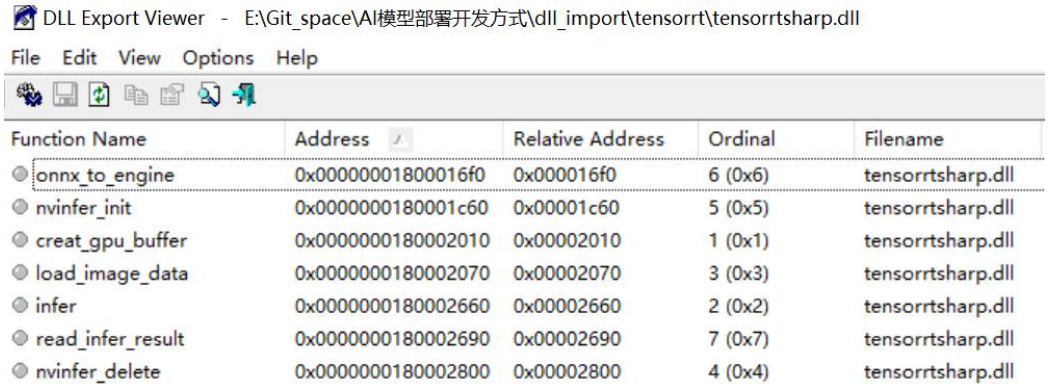
```
infer
read_infer_result
nvinfer_delete
```

LIBRARY 后所跟的为输出文件名，EXPORTS 后所跟的为输出方法名。仅需要以上语句便可以替代 extern "C" 、 __declspec(dllexport) 以及__stdcall 的使用。

5.2.5 生成 dll 文件

前面我们将项目配置输出设置为了生成 dll 文件，因此该项目不是可以执行的 exe 文件，只能生成不能运行。右键项目，选择重新生成/生成。在没有错误的情况下，会看到项目成功的提示。可以看到 dll 文件在解决方案同级目录下\x64\Release\文件夹下。

使用 dll 文件查看器打开 dll 文件，如图 5- 1 tensorrt.dll 文件方法输出目录所示；可以看到，我们创建的四个方法接口已经暴露在 dll 文件中。



Function Name	Address	Relative Address	Ordinal	Filename
onnx_to_engine	0x00000001800016f0	0x000016f0	6 (0x6)	tensorrtsharp.dll
nvinfer_init	0x0000000180001c60	0x00001c60	5 (0x5)	tensorrtsharp.dll
creat_gpu_buffer	0x0000000180002010	0x00002010	1 (0x1)	tensorrtsharp.dll
load_image_data	0x0000000180002070	0x00002070	3 (0x3)	tensorrtsharp.dll
infer	0x0000000180002660	0x00002660	2 (0x2)	tensorrtsharp.dll
read_infer_result	0x0000000180002690	0x00002690	7 (0x7)	tensorrtsharp.dll
nvinfer_delete	0x0000000180002800	0x00002800	4 (0x4)	tensorrtsharp.dll

图 5- 1 tensorrt.dll 文件方法输出目录

5.3 C#构建 Nvinfer 类

5.3.1 新建 C#类库

右击解决方案，添加->新建项目，选择添加 C#类库，项目名命名为 csharp_tensorrt_class，项目框架根据电脑中的框架选择，此处使用的是.NET 5.0。新建完成后，然后右击项目，选择添加->新建项，选择类文件，添加 Nvinfer.cs 和 NativeMethods.cs 两个类文件。

5.3.2 引入 dll 文件中的方法

在 NativeMethods.cs 文件下，我们通过[DllImport()]方法，将 dll 文件中所有的方法读取到 C#中。模型转换方法读取方式如下：

```
[DllImport(tensorrt_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
public extern static void onnx_to_engine(string onnx_file_path, string engine_file_path, int type);
```

其中 openvino_dll_path 为 dll 文件路径，CharSet = CharSet.Unicode 代表支持中文编码格式字符串，CallingConvention = CallingConvention.Cdecl 指示入口点的调用约定为调用方清理堆栈。

上述所列出的为初始化推理模型，dlii 文件接口在匹配时，是通过方法名字匹配的，因此，方法名要保证与 dll 文件中一致。其次就是方法的参数类型要进行对应，在上述方法中，函数的返回值在 C++中为 void* ，在 C#中对应的为 IntPtr 类型，输入参数中，在 C++

中为 wchar_t* 字符指针，在 C#中对应的为 string 字符串。通过方法名与参数类型一一对应，在 C#可以实现对方法的调用。其他方法在 C#重写后如下：

```
// 读取本地 engine 模型，并初始化 NvinferStruct
public extern static IntPtr nvinfer_init(string engine_filename, int num_ionode);
// 创建 GPU 显存输入/输出缓冲区
public extern static IntPtr creat_gpu_buffer(IntPtr nvinfer_ptr, string node_name, ulong data_length);
// 加载图片输入数据到缓冲区
public extern static IntPtr load_image_data(IntPtr nvinfer_ptr, string node_name, ref byte image_data, ulong image_size, int BN_means);
// 模型推理
public extern static IntPtr infer(IntPtr nvinfer_ptr);
// 读取推理数据
public extern static void read_infer_result(IntPtr nvinfer_ptr, string node_name_wchar, ref float result, ulong data_length);
// 删除内存地址
public extern static void nvinfer_delete(IntPtr nvinfer_ptr);
```

5.3.3 创建 Nvinfer 类

为了更方便地调用我们通过 dll 引入的 TensorRT 方法，减少使用时的函数方法接口，我们在 C#中重新组建我们自己的推理类，命名为 Class Nvinfer，其主要成员变量和方法如图 5- 2 Nvinfer 类图所示。

```
public class Nvinfer{}
```

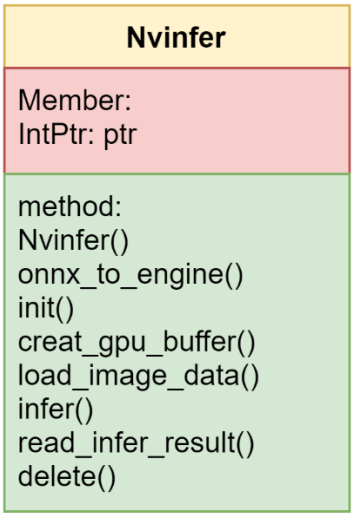


图 5- 2 Nvinfer 类图

在 Nvinfer 类中，我们只需要创建一个地址变量，作为 Nvinfer 类的成员变量，用于接收接口函数返回的推理核心指针，该成员变量我们只需要在当前类下访问，因此将其设置为私有变量：

```
private IntPtr ptr = new IntPtr();
```

首先封装模型转换方法 onnx_to_engine()，该方法主要用于将 onnx 模型转为 engine 格

式，因为 engine 为基于本机配置转换的推理模型文件，因此该模型文件不具备通用性，需要自行转换。在该方法中，只需要输入本地 onnx 模型文件、转换后的 engine 本地保存路径以及转换后的模型精度类型，通过调用重写的 NativeMethods.onnx_to_engine()方法即可。

```
public void onnx_to_engine(string onnx_file_path, string engine_file_path, AccuracyFlag type){
    NativeMethods.onnx_to_engine(onnx_file_path, engine_file_path, (int)type);
}
```

接下来，构建推理模型初始化方法 init()，我们只需要输入 engine 模型文件路径地址以及输入输出节点数量即可，然后调用 NativeMethods.nvinfer_init()方法，该方法可以实现本地读取 engine 模型，并初始化推理引擎结构体中的相关下成员变量。

```
public void init(string engine_filename, int num_ionode){
    ptr = NativeMethods.nvinfer_init(engine_filename, num_ionode);
}
```

creat_gpu_buffer()主要实现在 GPU 显存创建输入/输出缓冲区，此处需要指定输入/输出节点名以及输入输出节点数据大小。

```
public void creat_gpu_buffer(string node_name, ulong data_length){
    ptr = NativeMethods.creat_gpu_buffer(ptr, node_name, data_length);
}
```

load_image_data()该方法主要是是将带推理数据加载到推理模型中，该方法输入图片数据为转为矩阵的图片数据，方便图片数据在 C++与 C#之间进行传递，该方法中已经包括了图片数据预处理等步骤，因此在此处我们不需要再进行数据预处理。

```
public void load_image_data(string node_name, byte[] image_data, ulong image_size, BNFlag BN_means){
    ptr = NativeMethods.load_image_data(ptr, node_name, ref image_data[0], image_size, (int)BN_means);
}
```

infer()步骤主要是调用模型推理方法将配置好的数据进行模型推理。

```
public void infer(){
    ptr = NativeMethods.infer(ptr);
}
```

read_infer_result()主要实现了模型推理后推理结果数据的读取，目前对于结果的数据类型只支持浮点型数据的读取，后续如果有其他数据读取的要求，会根据需求进行更改。

```
public float[] read_infer_result(string node_name_wchar,ulong data_length){
    float[] result = new float[data_length];
    NativeMethods.read_infer_result(ptr, node_name_wchar, ref result[0], data_length);
    return result;
}
```

最后一步主要实现对内存数据的删除，放置占用太多的内存导致内存泄露。

```
public void delete(){
    NativeMethods.nvinfer_delete(ptr);
}
```

5.3.4 编译 Nvinfer 类库

右击项目，点击生成/重新生成，出现如图 5- 3 Nvinfer 类编译输出所示，表示编译成功。

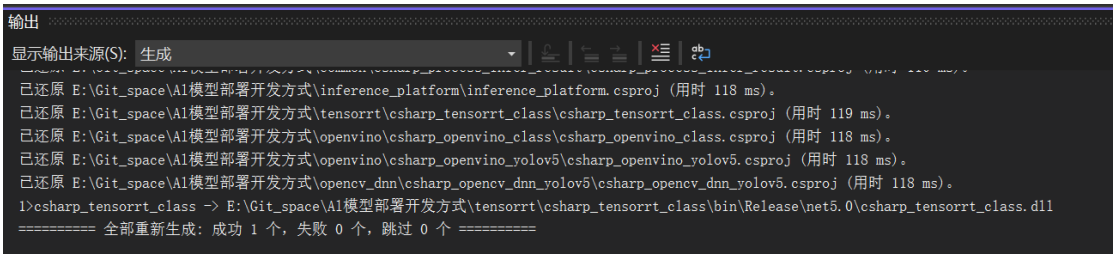


图 5- 3 Nvinfer 类编译输出

5.4 TensorRT 部署模型 C#实现

5.4.1 C#项目配置

在该项目中，我们需要添加 csharp_tensorrt_class 项目引用，以及 OpenCvsharp 程序包，具体操作方式参考 OpenVINOTM 部署模型 C#实现——C#项目配置。

5.4.2 TensorRT 部署 Yolov5 模型

（1）引入模型相关信息

```
string engine_path = "E:/Text_Model/yolov5/yolov5s1.engine";
string image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
string input_node_name = "images";
string output_node_name = "output";
```

（2）初始化推理核心类

由于有些情况下我们需要先一步进行模型转换，所以不能再模型初始化的时候直接读取转换好的模型文件，所以在此处多加一步进行模型的初始化。

```
// 创建模型推理类
Nvinfer nvinfer = new Nvinfer();
// 读取模型信息
nvinfer.init(engine_path, 2);
```

（3）配置输入输出 gpu 缓存区

在此处我们分别创建输入与输出节点的缓存区，并且需要指定输入与输出数据的数据长度。

```
nvinfer.creat_gpu_buffer(input_node_name, 640 * 640 * 3);
nvinfer.creat_gpu_buffer(output_node_name, 25200 * 85);
```

（4）加载推理图片数据

由于 Yolov5 模型已经固定输入数据大小，因此此处不需要在设置大小，根据 Yolov5 模型要求输入，进行图片预处理。

```
Mat image = new Mat(image_path);
```

```
// 将图片放在矩形背景下
Mat max_image = Mat.Zeros(new Size(1024, 1024), MatType.CV_8UC3);
Rect roi = new Rect(0, 0, image.Cols, image.Rows);
image.CopyTo(new Mat(max_image, roi));
```

此处通过 OpenCvSharp 将图片数据读取到内存中，并将图片放置在正方形背景下，后续的其他数据处理会在封装的 C++程序中实现，因此此处不需要再做进一步处理。接下来将图片转为矩阵数据，方便数据能在 C++与 C#之间传输。

```
byte[] image_data = new byte[2048 * 2048 * 3];
ulong image_size = new ulong();
image_data = max_image.ImEncode(".bmp");
image_size = Convert.ToUInt64(image_data.Length);
```

接下来直接调用 load_image_data ()加载输入数据方法，加载输入数据。

```
nvinfer.load_image_data(input_node_name, image_data, image_size, BNFlag.Normal);
```

（5）模型推理

```
nvinfer.infer();
```

（6）结果处理

首先是读取输出结果，yolov5 模型数据输出为浮点型数据，因此只需要调用 read_infer_result()方法读取数据即可。

```
float[] result_array = nvinfer.read_infer_result(output_node_name, 25200 * 85);
```

然后就是处理读取后的数据，在此处我们提供了一个专门针对 Yolov5 结果处理类，用于处理结果，此处我们只需要调用该类，便可以实现。

```
ResultYolov5 result = new ResultYolov5();
result.read_class_names(lable_path);
result.factor = (float)(image.Cols > image.Rows ? image.Cols : image.Rows) / (float)640;
Mat result_image = result.process_resule(image,result_array);
```

最终输出结果为处理完的结果图片，如图 4- 1 所示。

第6章 ONNX runtime 部署 AI 模型实现

6.1 ONNX runtime 部署模型 C++实现

6.1.1 部署工具配置

右击项目，点击管理 NuGet 程序包，在出现的页面，点击浏览，在搜索况中输入 onnx runtime，点击搜索后，会看到 Microsoft.ML.OnnxRuntime 程序包，如图 6- 1 所示。

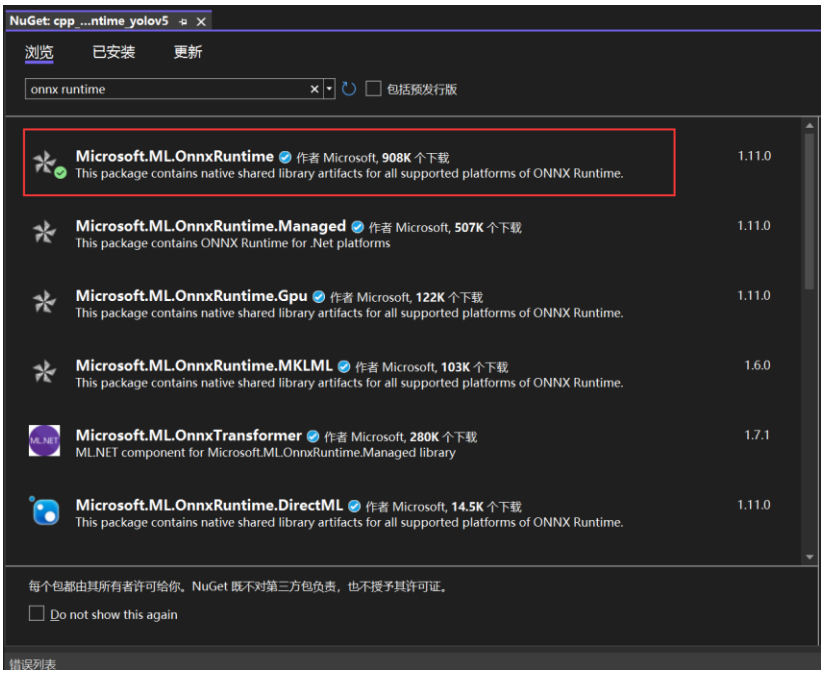


图 6- 1 onnx runtime 程序包

点击该程序包，会看到图 6- 2 所示的程序包信息，点击安装，将程序包安装到该项目下。

图 6- 2 onnx runtime 程序包详细信息

该程序包头文件为：<onnxruntime_cxx_api.h>，命名空间为 Ort。

6.1.2 部署 Yolov5 模型

（1）引入模型的相关信息

```
const char* model_path_onnx = "E:/Text_Model/yolov5/yolov5s.onnx";
const char* image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
std::string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
const char* input_node_name = "images";
const char* output_node_name = "output";
```

（2）设置推理环境

```
// 推理环境设置
Ort::Env env(ORT_LOGGING_LEVEL_VERBOSE, "OnnxModel");
// 设置推理会话
Ort::SessionOptions session_options;
// 设置线程数，使用 1 个线程执行，若想提升速度，增加线程数
session_options.SetIntraOpNumThreads(1);
session_options.SetGraphOptimizationLevel(GraphOptimizationLevel::ORT_ENABLE_ALL);
```

（3）初始化本地模型

初始化推理类，并直接读取本地模型。

```
Ort::Session session(env, multiByteToWideChar(model_path_onnx), session_options);
```

（4）配置模型输入信息

首先将输入输出节点放置到 vector 容器中；

```
// 输入/输出名字容器
// 将名字放在容器中输入数据时会更方便
std::vector<const char*> input_names{ input_node_name };
std::vector<const char*> output_names = { output_node_name };
```

接下来获取模型输入与输出的节点形状信息；

```
auto input_dims = session.GetInputTypeInfo(0).GetTensorTypeAndShapeInfo().GetShape();
auto output_dims = session.GetOutputTypeInfo(0).GetTensorTypeAndShapeInfo().GetShape();
```

0 为输入输出的节点序号，可以通过以下方法获取输出与输入节点数量；

```
size_t num_input_nodes = session.GetInputCount();
size_t num_output_nodes = session.GetOutputCount();
```

查询玩数量后，通过下面方式查询对应输出节点；

```
Ort::AllocatorWithDefaultOptions allocator;
const char* input_name = session.GetInputName(0, allocator);
const char* output_name = session.GetOutputName(0, allocator);
```

然后就是按照模型输入要求与处理数据，数据处理的方式与前面测试模型时处理方式基本相似；

```
cv::Mat image = cv::imread(image_path);
//将图片的放入到方型背景中
int max_side_length = std::max(image.cols, image.rows);
cv::Mat max_image = cv::Mat::zeros(cv::Size(max_side_length, max_side_length), CV_8UC3);
cv::Rect roi(0, 0, image.cols, image.rows);
image.copyTo(max_image(roi));
// 将图像归一化，转换 RGB 通道，并放缩到指定大小
cv::Mat normal_image = cv::dnn::blobFromImage(max_image, 1 / 255.0,
    cv::Size(input_dims[2], input_dims[3]), cv::Scalar(0, 0, 0), true, false);
```

最后是将数据转化为模型输入类型。ONNX runtime 要求的数据类型为 Ort::Value 类型，该型数据可以通过 CreateTensor()进行创建。

```
Ort::MemoryInfo memory_info = Ort::MemoryInfo::CreateCpu(OrtAllocatorType::OrtArenaAllocator,
    OrtMemType::OrtMemTypeDefault);
// 创建输入数据容器
std::vector<Ort::Value> input_tensors;
// 将输入数据加载到容器中
input_tensors.emplace_back(Ort::Value::CreateTensor<float>(memory_info, normal_image.ptr<float>(),
    normal_image.total(), input_dims.data(), input_dims.size()));
```

（5）模型推理

ONNX runtime 加载输入数据、模型推理以及读取推理结果数据合并在了一步中，通过以下方法进行实现：

```
std::vector<Ort::Value> output_tensors = session.Run(Ort::RunOptions{ nullptr }, input_names.data(),
    input_tensors.data(), input_names.size(), output_names.data(), output_names.size());
```

（6）推理结果处理

上一步我们读取出来的数据为 Ort::Value 数据类型，我们首先将其转化为普通的数组类型，然后再做进一步处理。

```
float* result_array = output_tensors[0].GetTensorMutableData<float>();
```

接下来直接调用 Yolov5 结果处理类，进行最后的结果处理。

```
ResultYolov5 result;
result.factor = max_side_length / (float)640;
result.read_class_names(lable_path);
// 处理输出结果
cv::Mat result_image = result.yolov5_result(image, result_array);
```

6.2 ONNX runtime 部署模型 C#实现

6.2.1 C#环境设置

ONNX runtime 环境设置与 C++中实现一致，都是通过 NuGet 程序包安装，具体可以参考上一步中 C++环境配置；除此以外，此处我们还需要使用 OpenCvsharp，所以还需要安装 OpenCvsharp4.

6.2.2 部署 Yolov5 模型

（1）引入模型相关信息

```
string model_path = "E:/Text_Model/yolov5/yolov5s.onnx";
string image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
string input_node_name = "images";
string output_node_name = "output";
```

（2）创建会话与基本环境设置

```
SessionOptions options = new SessionOptions();
options.LogSeverityLevel = OrtLoggingLevel.ORT_LOGGING_LEVEL_INFO;
// 设置为 CPU 上运行
options.AppendExecutionProvider_CPU(0);
```

（3）初始化模型推理

```
InferenceSession onnx_session = new InferenceSession(model_path, options);
```

（4）配置模型输入

首先是预处理数据，在此处我们先进行数据的除归一化之外其他预处理，将归一化放在创建推理数据中。

```
Mat image = new Mat(image_path);
// 将图片放在矩形背景下
Mat max_image = Mat.Zeros(new Size(1024, 1024), MatType.CV_8UC3);
Rect roi = new Rect(0, 0, image.Cols, image.Rows);
image.CopyTo(new Mat(max_image, roi));
// 将图片转为 RGB 通道
Mat image_rgb = new Mat();
```

```
Cv2.CvtColor(max_image, image_rgb, ColorConversionCodes.BGR2RGB);
Mat resize_image = new Mat();
Cv2.Resize(image_rgb, resize_image, new Size(640, 640));
```

创建 Tensor 数据，Tensor 为 ONNX runtime 模型推理的数据类型，此处如果是多输入的模型，就要分开创建。

```
Tensor<float> input_tensor = new DenseTensor<float>(new[] { 1, 3, 640, 640 });
for (int y = 0; y < resize_image.Height; y++){
    for (int x = 0; x < resize_image.Width; x++) {
        input_tensor[0, 0, y, x] = resize_image.At<Vec3b>(y, x)[0] / 255f;
        input_tensor[0, 1, y, x] = resize_image.At<Vec3b>(y, x)[1] / 255f;
        input_tensor[0, 2, y, x] = resize_image.At<Vec3b>(y, x)[2] / 255f;
    }
}
```

最后就是创建输入容器，然后将输入数据放置在输入容器中，并制定输入节点名称，此处是通过节点名称指定节点输入的。

```
List<NamedOnnxValue> input_ontainer = new List<NamedOnnxValue>();
input_ontainer.Add(NamedOnnxValue.CreateFromTensor(input_node_name, input_tensor));
```

（5）模型推理

```
IDisposableReadOnlyCollection<DisposableNamedOnnxValue> result_infer = onnx_session.Run(input_ontainer);
```

（6）读取模型结果

上一步我们在模型推理时已经将结果数据放在了 result_infer 中，不过该数据类型我们不能直接读取使用，通过以下转换，将其转为 float 数组。

```
DisposableNamedOnnxValue[] results_array = result_infer.ToArray();
// 读取第一个节点输出并转为 Tensor 数据
Tensor<float> tensors = results_array[0].AsTensor<float>();
// 将数据转为 float 数组
float[] result_array = tensors.ToArray();
```

然后就是处理读取后的数据，在此处我们提供了一个专门针对 Yolov5 结果处理类，用于处理结果，此处我们只需要调用该类，便可以实现。

```
ResultYolov5 result = new ResultYolov5();
result.read_class_names(lable_path);
result.factor = (float)(image.Cols > image.Rows ? image.Cols : image.Rows) / (float)640;
Mat result_image = result.process_resule(image,result_array);
```

最终输出结果为处理完的结果图片。

第7章 OpenCV Dnn 部署 AI 模型实现

7.1 OpenCV Dnn 部署模型 C++实现

7.1.1 C++环境配置

OpenCV Dnn 为 OpenCV 下的一个用于部署深度学习模型的库，配置 OpenCV 的时候已经引入了 Dnn 的库。

7.1.2 部署 Yolov5 模型

（1）引入模型相关信息

```
const char* model_path_onnx = "E:/Text_Model/yolov5/yolov5s.onnx";
const char* model_path_engine = "E:/Text_Model/yolov5/yolov5s.engine";
const char* image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
std::string lable_path = "E:/Git_space/AI 模型部署开发方式/model/yolov5/lable.txt";
const char* input_node_name = "images";
const char* output_node_name = "output";
```

（2）初始化网络、读取本地模型信息

OpenCV Dnn 支持多种模型格式文件的读取，针对不同的模型使用不同的读取方法。

```
cv::dnn::Net net = cv::dnn::readNetFromONNX(model_path_onnx);
```

（3）配置模型输入

此处的模型输入数据处理与前面相似，不在做过多解释

```
// 图象预处理 - 格式化操作
cv::Mat image = cv::imread(image_path);
// 将图片数据放置在方形背景上
int max_side_length = std::max(image.cols, image.rows);
cv::Mat max_image = cv::Mat::zeros(cv::Size(max_side_length, max_side_length), CV_8UC3);
cv::Rect roi(0, 0, image.cols, image.rows);
image.copyTo(max_image(roi));
// 将图像归一化，并放缩到指定大小
cv::Size input_node_shape(640, 640);
cv::Mat BN_image = cv::dnn::blobFromImage(max_image, 1 / 255.0, input_node_shape, cv::Scalar(0, 0, 0), true, false);
```

接下来就是将处理完的数据加载到模型上，只需要调用 setInput()方法即可，该方法可以指定模型名字进行数据加载。

```
net.setInput(BN_image, input_node_name);
```

（4）模型推理

```
cv::Mat result_mat= net.forward();
```

（5）推理结果处理

上一步在模型推理后，已将将推理结果存储到 Mat 数据中，我们首先将 Mat 数据转为数组数据。

```
float* result_array = (float*)result_mat.data;
```

接下来直接调用 Yolov5 结果处理类，进行最后的结果处理。

```
ResultYolov5 result;
result.factor = max_side_length / (float)640;
result.read_class_names(lable_path);
// 处理输出结果
cv::Mat result_image = result.yolov5_result(image, result_array);
```

7.2 OpenCV Dnn 部署模型 C++实现

7.2.1 C++环境配置

OpenCV Dnn 为 OpenCV 下的一个用于部署深度学习模型的库，所以在此处只需要通过 NuGet 功能，安装 OpenCvSharp4 即可。

7.2.2 部署 Yolov5 模型

（1）引入模型相关信息

```
string model_path = "E:/Text_Model/yolov5/yolov5s.onnx";
string image_path = "E:/Text_dataset/YOLOv5/0001.jpg";
string lable_path = "E:/Git_space/Al 模型部署开发方式/model/yolov5/lable.txt";
string input_node_name = "images";
string output_node_name = "output";
```

（2）初始化网络、读取本地模型信息

OpenCV Dnn 支持多种模型格式文件的读取，针对不同的模型使用不同的读取方法。

```
Net net = CvDnn.ReadNetFromOnnx(model_path);
```

（3）配置模型输入

此处的模型输入数据处理与前面相似，不在做过多解释

```
Mat image = new Mat(image_path);
// 将图片放在矩形背景下
Mat max_image = Mat.Zeros(new Size(1024, 1024), MatType.CV_8UC3);
Rect roi = new Rect(0, 0, image.Cols, image.Rows);
image.CopyTo(new Mat(max_image, roi));
// 数据归一化处理
Mat BN_image = CvDnn.BlobFromImage(max_image, 1 / 255.0, new Size(640, 640), new Scalar(0, 0, 0), true, false);
```

接下来就是将处理完的数据加载到模型上，只需要调用 setInput()方法即可，该方法可以指定模型名字进行数据加载。

```
net.SetInput(BN_image);
```

（4）模型推理

```
Mat result_mat = net.Forward();
```

（5）推理结果处理

上一步在模型推理后，已将将推理结果存储到 Mat 数据中，我们首先将 Mat 数据转为数组数据。

```
Mat result_mat_to_float = new Mat(25200, 85, MatType.CV_32F, result_mat.Data);
float[] result_array = new float[result_mat.Cols* result_mat.Rows];
```

接下来直接调用 Yolov5 结果处理类，进行最后的结果处理。

```
ResultYolov5 result = new ResultYolov5();
// 读取本地模型类别信息
result.read_class_names(lable_path);
// 图片加载缩放比例
result.factor = (float)(image.Cols > image.Rows ? image.Cols : image.Rows) / (float)640;
// 处理输出数据
Mat result_image = result.process_resule(image, result_array);
```

第8章 AI 模型部署平台软件开发

8.1 Winform 环境搭建

8.1.1 新建窗体项目

右击解决方案，选择新建项目，然后选择 Windows 窗体应用，如图 8 - 1 所示。

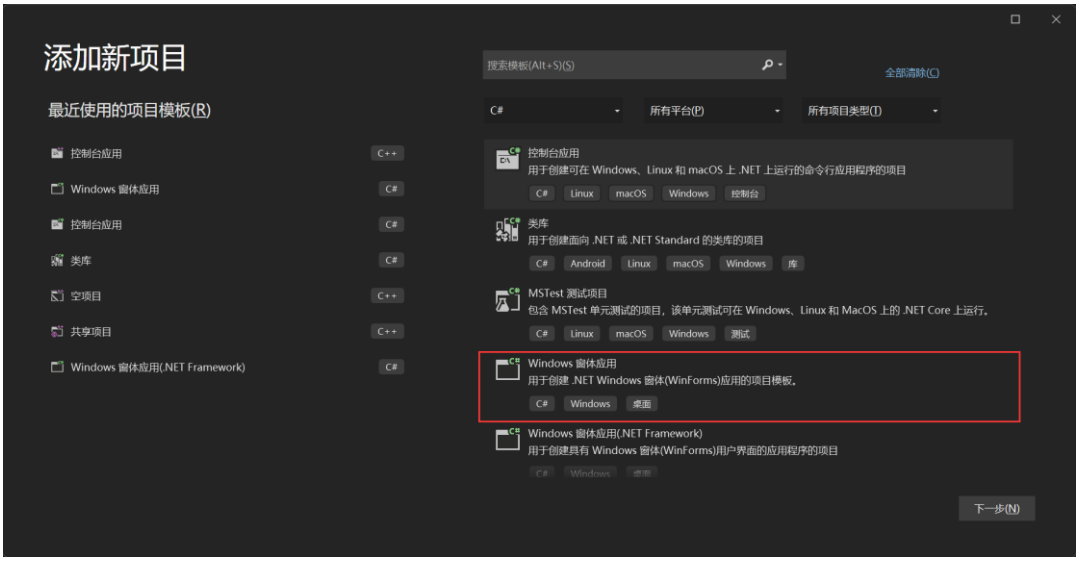


图 8 - 1 新建 Windows 窗体应用

项目框架可以根据自己电脑安装框架进行选择，此处选择为 Net5.0 框架。

8.1.2 添加项目引用

右击该项目，选择添加，项目引用，在此处我们需要添加我们创建的 OpenVinoSharp 下的 Core 类、TensorRTSharp 下的 Nvinfer 类以及推理结果处理类，如图 8 - 2 所示。

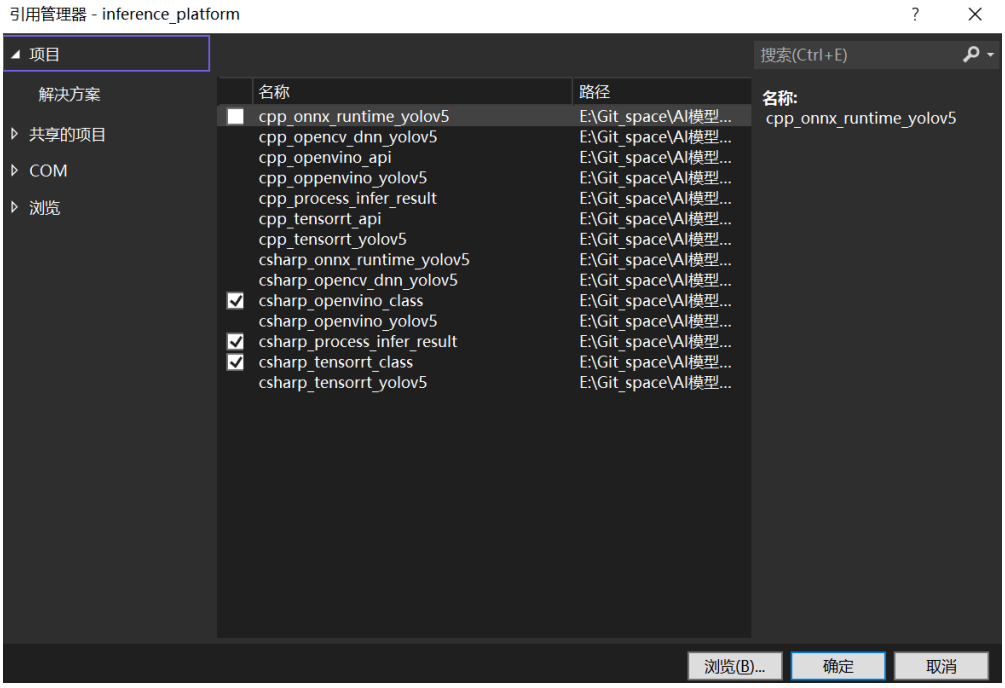


图 8 - 2 项目引用选择

8.1.3 添加程序包

此处我们需要添加 OpenCvSharp 以及 ONNX runtime 的程序包，我们通过 VS 中的 NuGet 程序功能添加，具体操作参考 ONNX runtime 部署模型 C++实现中 C++环境设置。

8.2 软件操作页面说明

针对推理平台要求的功能，设计了如图 8 - 3 所示页面，其重要分为 8 各区域。

图 8 - 3 推理平台页面

区域 1 为模型选择区域，针对现有可利用模型，可以使用 Yolov5 以及 ResNet50 两种模型，针对其他模型，目前还在测试中；区域 2 为模型文件选择区域，由于不同推理平台所要求的模型文件类型不一致，因此要根据选用的测试平台来选择测试文件；区域 3 为部署平台选择，该测试平台支持 OpenVINO、TensorRT、ONNX runtime 以及 OpenCV Dnn 平台；区域 4 为测试图片以及 lable 文件选择，测试图片为用于推理测试的图片，lable 文件主要是用于推理结果的处理时分类结果查询；区域 5 为模型测试按键，此处主要用于模型部署测试，模型仅推理一次；区域 6 为模型推理时间测试，在此处需要设置测试次数，最终结果为 n 次推理后的平均值；区域 7 为推理结果图片输出；区域 8 为推理信息打印输出。

第9章 AI 模型部署平台对比

9.1 基本信息对比

	OpenVINO	TensorRT	ONNX runtime	OpenCV Dnn
推出厂商	因特尔（Intel）	英伟达（NVIDIA）	微软（Microsoft）	OpenCV
套件工具	Model Optimizer、Inference Engine			
是否开源	开源、商用免费	开源	开源	开源
编程语言	C、C++	C++		C、C++
支持操作系统	Linux、Windows、macOS、Raspbian	Linux、Windows、macOS、Android	Linux、Windows、macOS	Linux、Windows、macOS
程序接口	C++ API Python API	C++ API Python API	C++ API Python API C# API	C++ API Python API C# API
硬件支持	英特尔自家厂商推出的 CPU、GPU、FPGA、VPU、以及其他厂商推出的处理器设备	英伟达自家厂商推出的 GPU 平台		
支持深度学习框架	TensorFlow、Caffe、Mxnet、Pytorch、ONNX	TensorFlow、Caffe、Mxnet、Pytorch、ONNX	Caffe2, PyTorch, MXNet，ML.NET，TensorRT、Microsoft CNTK	Darknet、Caffe、TensorFlow、Torch7、ONNX、

9.2 API 接口对比

对于该项目中所列举使用的四种模型部署工具，由于支持语言不同，我们此处选择共同的支持语言 C++API 接口进行对比。

推理过程	推理套件	C++ API 接口
推理引擎类	OpenVINO	Ov::Core
	TensorRT	nvinfer1::ICudaEngine
	ONNX runtime	Ort::Session
	OpenCV Dnn	cv::dnn::Net

模型读取	OpenVINO	std::shared_ptr<ov::Model> model_ptr = core.read_model(model_path);
	TensorRT	普通二进制数据文件读取
	ONNX runtime	加载时同步读取
	OpenCV Dnn	加载时同步读取
模型加载	OpenVINO	ov::CompiledModel compiled_model = core.compile_model(model_ptr, "CPU");
	TensorRT	ICudaEngine* deserializeCudaEngine()
	ONNX runtime	Ort::Session session()
	OpenCV Dnn	Net readNetFromONNX()
创建推理通道	OpenVINO	ov::InferRequest infer_request = compiled_model.create_infer_request();
	TensorRT	IExecutionContext* createExecutionContext();
	ONNX runtime	单通道推理
	OpenCV Dnn	单通道推理
推理数据加载	OpenVINO	fill_tensor_data_image(input_image_tensor, normal_image);
	TensorRT	
	ONNX runtime	std::vector<Ort::Value>数据容器
	OpenCV Dnn	net.setInput(, input_node_name);
模型推理	OpenVINO	infer_request.infer();
	TensorRT	
	ONNX runtime	
	OpenCV Dnn	
推理结果读取	OpenVINO	float* result_array = output_tensor.data<float>();
	TensorRT	
	ONNX runtime	
	OpenCV Dnn	

9.3 推理时间对比

为了对比各个平台推理时间，我们针对四个平台基于 ResNet50 以及 Yolov5s 网络进行了部署推理，基于我们前期搭建的测试平台实现，测试轮次为 100 次。测试结果如表 9 - 1 所示。

表 9 - 1 不同平台模型推理时间

测试环境	部署平台	程序各过程运行时间/(ms)

		总时间	前处理	推理运算	后处理
Shape：[1, 3, 224 ,224]	OpenVINO	313.20	288.20	24.56	0.44
Net：ResNet50	TensorRT	67.23	56.46	6.38	4.39
CPU：AMD R7 5800H	ONNX runtime	321.65	290.02	31.25	0.37
GPU：RTX 3060	OpenCV Dnn	257.86	153.13	104.06	0.66
Shape：[1, 3, 640 ,640]	OpenVINO	212.47	150.73	56.47	5.26
Net：Yolov5s	TensorRT	90.38	51.84	22.81	15.72
CPU：AMD R7 5800H	ONNX runtime	229.57	63.66	162.25	3.65
GPU：RTX 3060	OpenCV Dnn	180.52	43.47	132.39	4.66

基于表 9 - 1，我们可以看出，四种模型部署平台中，OpenVINO 与 TensorRT 表现出了优良的模型推理性能，其中 OpenVINO 的部署与运行是在 CPU 平台上实现，TensorRT 的部署时在 GPU 上实现的。

第10章 模型量化实现

10.1 模型量化

这些年来，深度学习在众多领域的表现使其成为了如今机器学习的主流方向，但其巨大的计算量严重影响了模型推理的速度；并且随着越来越多基于深度神经网络在端侧上的智能应用，深度神经网络巨大计算量成为了部署模型时的巨大挑战。究其原因，主要是模型训练时为了保证较高的精度，大部分的运算都是采用浮点型进行计算，常见的是 32 位浮点型和 64 位浮点型，即 float32 和 double64。但模型推理没有反向传播，网络中存在很多不重要的参数，或者并不需要太细的精度来表示它们。为了弥补端侧智能应用的算力需求与端侧的算力能力的鸿沟，近几年来模型压缩成为了业界的热点之一。而模型量化属于非常实用的模型压缩技术，并且当前已经在工业界发展非常成熟。

模型量化就是将训练好的深度神经网络的权值、激活值等从高精度转化成低精度的操作过程，例如将 32 位浮点数转化成 8 位整型数 int8，同时我们转换后的模型准确率与转化前相近。在经过模型量化后，可以减少内存和存储占用，降低功耗以及提升运算速度。

10.2 ResNet50 量化 OpenVINO™ 实现

OpenVINO™ 提供了专门的模型量化工具 pot 工具包，以及模型精度检查工具 accuracy_checker 工具包，不过该工具包在使用时会出现一些问题，所以我们在此处结合 OpenVINO™ 官方提供的案例，实现 ResNet50 量化。

10.2.1 模型转换

如果我们拿到的模型为 onnx 格式，我们首先要将模型转为 IR 格式才可以进行下一步，使用的 mo 工具就可以实现。

首先打开 openvino 虚拟环境，切换到 tools 目录下

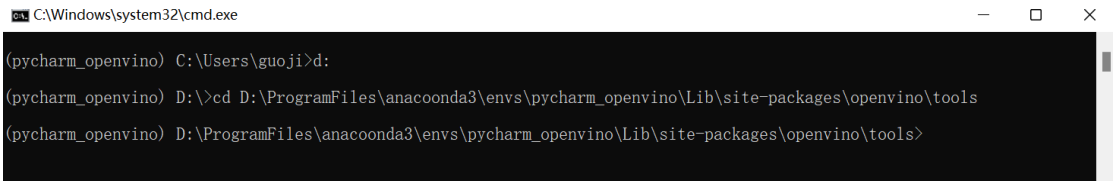


图 10- 1 命令运行窗口

接下来输入以下命令：

```
mo --framework=onnx --data_type=FP16 --input_shape=[1,3,224,224]
-m E:\Text_Model\flowerclas20220427\flower_clas.onnx
--output_dir E:\Text_Model\flowerclas20220427
```

其中 framework 指的是输入模型格式为 onnx 格式，data_type 指的是模型输出精度，-m 指的是输入模型的路径地址，output_dir 指的是输出模型位置

运行最后输出如图 10- 2 所示，在指定文件夹中可以查看到转换成功的对应文件。

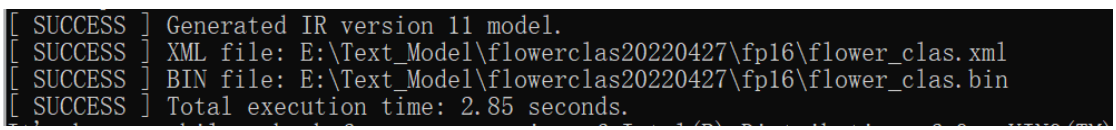


图 10- 2 IR 模型转换结果

10.2.2 定义数据集加载类

本次转换过程通过 python 实现，首先我们先编写本地数据集读取以及预处理类，新建 dataloader.py 文件。

首先引入相关程序包：

```
from openvino.tools.pot.api import DataLoader, Metric
import cv2
```

然后定义数据集加载类：

```
class DataLoader(DataLoader):
```

接下来定义一下相关的初始化函数：

```
def __init__(self, config):
        path = config['data_path']
        file = config['data_file']
        self.indexes, self.pictures, self.labels = self.load_data(path,file)

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

def __getitem__(self, index):
        if index >= len(self):
            raise IndexError
        return (self.indexes[index], self.labels[index]), self.pictures[index]
```

此处主要是初始化函数，初始化函数返回的值为序列号、图片数据以及图片标签列表，主要通过调用 load_data()成员方法实现。load_data()成员方法为：

```
def load_data(self, path,file):
       pictures, pictures_path, labels, indexes = [], [], [],[]
       with open(path+file, 'r') as f:
            for line in f:
               line = line.split()
               pictures_path.append(line[0])
               labels.append(int(line[1]))
        idx = 0
        for pic_path in pictures_path:
            src = cv2.imread(path+pic_path)
            image = cv2.cvtColor(src, cv2.COLOR_BGR2RGB)
            image = image/255.0
            image -= [0.485, 0.456, 0.406]
            image /= [0.229, 0.224, 0.225]
            image = cv2.resize(image, (224, 224), interpolation=cv2.INTER_AREA)   # [82 202 255]>[51 228 254]
             image = image.transpose(2, 0, 1)
             pictures.append(image)
             indexes.append(idx)
             idx = idx+1
```

```
    return indexes, pictures, labels
```

该方法主要是读取本地数据集信息到内存，将标签数据转为整型数据存放；图片数据读取后需要按照模型要求，对图片数据进行预处理，最后转为矩阵数组放在列表中。**对于不同的数据集，需要对此处做相应的修改。**

10.2.3 定义精度设置类

新建 accuracy.py 文件，首先引入相关程序包：

```
import numpy as np
from openvino.tools.pot.api import DataLoader, Metric
```

然后就是定义精度类：

```
class Accuracy(Metric):

    # Required methods
    def __init__(self, top_k=1):
        super().__init__()
        self._top_k = top_k
        self._name = 'accuracy@top{}'.format(self._top_k)
        self._matches = []

    @property
    def value(self):
        """ Returns accuracy metric value for the last model output. """
        return {self._name: self._matches[-1]}

    @property
    def avg_value(self):
        return {self._name: np.ravel(self._matches).mean()}

    def update(self, output, target):
        """ Updates prediction matches.
        :param output: model output
        :param target: annotations
        """
        if len(output) > 1:
            raise Exception('The accuracy metric cannot be calculated '
                            'for a model with multiple outputs')
        if isinstance(target, dict):
            target = list(target.values())
        predictions = np.argsort(output[0], axis=1)[:, -self._top_k:]
        match = [float(t in predictions[i]) for i, t in enumerate(target)]

        self._matches.append(match)
```

```
    def reset(self):
        """ Resets collected matches """
        self._matches = []

    def get_attributes(self):
        return {self._name: {'direction': 'higher-better',
                             'type': 'accuracy'}}
```

对于这个精度设置类，我们在此处不需要做相应修改。

10.2.4 模型量化基本步骤

模型量化主要分为九个步骤：

（1）加载模型

首先第一步就是读取本地模型，此处读取的为 IR 格式的模型，因此要同时加载模型文件以及权重文件。

```
model = load_model(model_config)
```

在此处我们将模型文件本地地址放在字典中，直接使用 load_model()方法读取即可。

（2）初始化数据集

在前面我们定义了数据集加载类，因此在此处我们只需要调用该类即可：

```
data_loader = DataLoader(dataset_config)
```

对于数据集的路径信息，我们也是放在字典中，主要是定义的一个路径信息以及标签文件。

其中标签文件内容可以参考图 10- 3。

```
jpg/image_05101.jpg 1
jpg/image_05114.jpg 1
jpg/image_05100.jpg 1
jpg/image_05123.jpg 1
jpg/image_05120.jpg 1
jpg/image_05118.jpg 1
jpg/image_05116.jpg 1
jpg/image_05132.jpg 1
jpg/image_05104.jpg 1
jpg/image_06630.jpg 2
```

图 10- 3 标签文件

（3）初始化精度

精度初始化直接调用 Accuracy()类即可。

```
metric = Accuracy(top_k=1)
```

（4）初始化引擎

初始化用于度量计算和统计信息收集的引擎。

```
engine = IEEngine(engine_config, data_loader, metric)
```

engine_config 为设备信息设置，格式为字典类型。

```
engine_config = Dict({
    'device': 'CPU',
```

```
    'stat_requests_number': 2,
    'eval_requests_number': 2
})
```

（5）创建压缩运行通道

```
pipeline = create_pipeline(algorithms, engine)
```

algorithms 此处定义的为算法压缩的相关信息。

```
algorithms = [
    {
        'name': 'DefaultQuantization',
        'params': {
            'target_device': 'CPU',
            'preset': 'performance',
            'stat_subset_size': 300
        }
    }
]
```

（6）运行模型

```
compressed_model = pipeline.run(model)
```

（7）压缩权重

压缩量化精度的模型权重，以减小最终结果 bin 文件的大小。

```
compress_model_weights(compressed_model)
```

（8）保存模型到本地

```
compressed_model_paths = save_model(model=compressed_model,
save_path="E:\\Text_Model\\flowerclas20220427\\", model_name="flower_clas_quantized")
compressed_model_xml = compressed_model_paths[0]["model"]
compressed_model_bin = Path(compressed_model_paths[0]["model"]).with_suffix(".bin")
```

（9）精度检测

Pipeline 下的 evaluate()方法可以对模型的精度进行检验，直接将模型加载到上面即可，不过要与对应的数据集匹配

```
metric_results = pipeline.evaluate(model)
```

10.2.5 ResNet50 模型量化

首先导入相关程序集：

```
from pathlib import Path
from addict import Dict
from openvino.tools.pot.engines.ie_engine import IEEngine
from openvino.tools.pot.graph import load_model, save_model
from openvino.tools.pot.graph.model_utils import compress_model_weights
from openvino.tools.pot.pipeline.initializer import create_pipeline
from accuracy import Accuracy
from dataloader import DataLoader
```

定义本地文件相关信息，主要定义数据集信息与模型文件信息，使用字典功能实现：

```
model_config = Dict({
    'model_name': 'flower_clas',
    'model': "E:\\Text_Model\\flowerclas20220427\\flower_clas.xml",
    'weights': "E:\\Text_Model\\flowerclas20220427\\flower_clas.bin"
})
dataset_config = {
    'data_path': "E:\\Text_dataset\\flowers102\\",
    'data_file': "val_list.txt"
}
```

除此以外还需要定义设备信息 engine_config 以及压缩设置 algorithms。

接下来就是按照上面 9 个步骤进行模型量化：

```
def resnet50_to_int8():
    # Step 1: Load the model.
    model = load_model(model_config)
    print("Load the model.")

    # Step 2: Initialize the data loader.
    data_loader = DataLoader(dataset_config)
    print("Initialize the data loader.")

    # Step 3 (Optional. Required for AccuracyAwareQuantization): Initialize the metric.
    metric = Accuracy(top_k=1)
    print("Initialize the metric.")

    # Step 4: Initialize the engine for metric calculation and statistics collection.
    engine = IEEngine(engine_config, data_loader, metric)
    print("Initialize the engine for metric calculation and statistics collection.")

    # Step 5: Create a pipeline of compression algorithms.
    pipeline = create_pipeline(algorithms, engine)
    print("Create a pipeline of compression algorithms.")

    # Step 6: Execute the pipeline.
    compressed_model = pipeline.run(model)
    print("Execute the pipeline.")

    # Step 7 (Optional): Compress model weights quantized precision
    #                    in order to reduce the size of final .bin file.
    compress_model_weights(compressed_model)
    print("Compress model weights quantized precision")

    # Step 8: Save the compressed model to the desired path.
```

```
    compressed_model_paths = save_model(model=compressed_model,
save_path="E:\\Text_Model\\flowerclas20220427\\", model_name="flower_clas_quantized")
    compressed_model_xml = compressed_model_paths[0]["model"]
    compressed_model_bin = Path(compressed_model_paths[0]["model"]).with_suffix(".bin")
    print("Save the compressed model to the desired path.")

    # Step 9: Compare accuracy of the original and quantized models.
    metric_results = pipeline.evaluate(model)
    if metric_results:
        for name, value in metric_results.items():
            print(f"Accuracy of the original model: {name}: {value}")

    metric_results = pipeline.evaluate(compressed_model)
    if metric_results:
        for name, value in metric_results.items():
            print(f"Accuracy of the optimized model: {name}: {value}")

    print("Compare accuracy of the original and quantized models.")
```

为了更好地跟踪每一步运行情况，在每一步运行后都做了输出，便于观察程序运行情况。

最后我们在主函数中运行 resnet50_to_int8()方法即可。最终运行输出如图 10- 4 所示。

```
PS E:\Git_space\AI模型部署开发方式\quantification\openvino\resnet50_quantificatuon_int8> conda activate pycharm_openvino
PS E:\Git_space\AI模型部署开发方式\quantification\openvino\resnet50_quantificatuon_int8> & D:/ProgramFiles/anacoonda3/envs/
pycharm_openvino/python.exe e:/Git_space/AI模型部署开发方式/quantification/openvino/resnet50_quantificatuon_int8/main.py
Load the model.
Initialize the data loader.
Initialize the metric.
Initialize the engine for metric calculation and statistics collection.
Create a pipeline of compression algorithms.
Execute the pipeline.
Compress model weights quantized precision
Save the compressed model to the desired path.
Accuracy of the original model: accuracy@top1: 0.9852941176470589
Accuracy of the optimized model: accuracy@top1: 0.9852941176470589
Compare accuracy of the original and quantized models.
PS E:\Git_space\AI模型部署开发方式\quantification\openvino\resnet50_quantificatuon_int8>
```

图 10- 4 程序运行输出

在最后一步中，我们对原始模型以及转换后的模型进行了精度检测，原始模型与转换后模型测试精度均为 0.985，

最后我们对比一下模型推理时间。在同样的设备上，测试该模型推理所用时间，为了避免测试的偶然性，我们运行 100 次后求取平均运行时间，测试结果如表 10- 1 所示。

表 10- 1 模型推理时间对比

模型信息	设备信息	是否量化	程序各过程运行时间/(ms)			
			总时间	前处理	推理运算	后处理
Shape：[1, 3, 224 ,224]	CPU：AMD R7 5800H	是	192.10	173.61	18.03	0.45
Net：ResNet50	GPU：RTX 3060	否	355.80	330.84	24.49	0.49

通过对比模型量化前后的推理时间，我们可以看出，模型在量化后，读取模型时间大大减少，读取速度提升了 47.5%；另一方面，模型推理时间也有了明显提升。

10.3 总结

本节主要借助 OpenVINO™ 自带的模型量化算法，实现了将 ResNet50 模型有 F16 量化到 INT8，使得模型在损失很小的精度前提下，实现了模型推理速度的大幅度提升。

通过模型量化，减少了模型内存的占用，并且很大程度上提升模型推理的速度，这对模型部署在工业边缘设备以及小型设备等硬件上具有很大的推动作用。