南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

智能计算系统期末实验报告

模型部署与转换

姓名：宋佳蓁 管昀玫

学号：2013904 2013750

年级：2020 级

专业：计算机科学与技术

2023 年 6 月 20 日

摘要

关键字：Parallel

目录

一、 背景调研

(一) 深度学习框架介绍

近年来，随着人工智能快速发展，深度学习技术已经在许多领域发挥出巨大的作用，各种人工智能框架层出不穷，如 PyTorch、Tensorflow、Caffe、MXNet 等等，每个深度学习框架都有着各自的优缺点。如：

1. TensorFlow 是使用 c++ 语言开发设计的采用数据流图进行运算的开源软件库，其优点在于具有很好的灵活性、可移植性和通用性；缺点在于版本复杂、各版本之间的兼容性较差、底层接口可能会显得繁琐等。[1]

2. Pytorch 是一个由 Facebook 开发的深度学习框架，它以动态图形式演示模型的构建，以及实现和调试，支持大部分 Python 语法。其优点在于可以方便地与其它科学计算库（如 NumPy）结合使用、代码很简洁、易于使用、灵活性高；缺点在于运行效率低、部署复杂。[3]

3. MXNet 采用类似于 TensorFlow 的数据流图进行运算，支持多 GPU 配置，提供多语言接口，其优点在于灵活性高、运行速度快、支持多种设备；缺点在于框架的安装和调试难度较高、学习社区小、学习复杂度高。[4]

4. PaddlePaddle 是百度自主研发的集深度学习核心框架，有全面的官方支持的工业级应用模型，其优点在于代码干净、上手简单、内存占用较少；缺点在于框架的安装难度较高、学习文档较少。[5]

5. Caffe 由 Jiaxiang Wu 等人开发，是一个使用配置文件的形式定义网络结构的开源深度学习框架。Caffe 使用了一种称为”caffe.proto” 的配置文件来定义模型和训练参数。它的优点在于模块化、稳定性、迁移性较好、速度较快，支持 CPU 和 GPU 加速；缺点在于框架的安装和调试难度较高、各版本之间的兼容性较差、学习文档较少、学习成本较高。[2]

6. Darknet 由 Joseph Redmon 开发，是一个用 C 语言编写的开源框架，被广泛用于 YOLO（You Only Look Once）目标检测算法。Darknet 中包含了用于构建神经网络模型的基本层（如卷积、池化、全连接等）和常用函数（如非极大值抑制等）。其优点在于简洁、高效、易于安装，并支持 CPU 和 GPU 计算；缺点在于学习文档较少、学习社区较小。[9]

基于上述各种深度学习框架的优缺点的分析，不难想到根据实际应用或再训练场景的改变，开发者往往需要将训练好的深度学习模型部署到不同的学习框架中，因此不同框架下模型的互相转换变得非常重要。为了实现将某一个深度学习框架下训练好的模型部署到其他框架中这一任务，如果仅仅“就事论事”，让开发人员在另一框架中重新搭建相同结构的网络，整个部署过程的难度、时间开销极高、工程量极大且复用率极低。因此，如何进行通用性的模型转换并开发对应的模型转换部署工具变得格外重要。这种通用性的模型转换工具可以使开发人员直接得到目标框架下的神经网络模型，有效降低了神经网络模型部署所需要的时间成本及资源消耗，具有很强的实际应用意义。

(二) 模型转换部署技术

目前深度学习框架模型转换技术采用的设计方式主要有两种：第一种是直接转换技术 [6]，即将当前框架下的模型直接转换为目标框架下的模型

第二种设计思路是为深度学习专门设计一种开放式的模型文件保存规范 [7]，这种设计思路的代表是开放式神经网络交换格式 ONNX。ONNX 为网络模型提供了开源文件的保存格式，定义了具有可扩展性的计算图模型、标准数据类型以及内置运算符。

1. 直接转换技术

直接转换技术的转换过程可以划分为以下几个步骤：第一步是读取现有框架下的模型文件，分析识别内部张量数据、运算单元和计算图等信息；第二步是将识别到的模型结构和参数信息翻译成目标模型所需要的代码格式；第三步是使用目标框架保存转换后的模型文件 [8]。

直接转换技术简化了不同框架下的模型转换部署流程，具有良好的效果，而由个人、企业在深度学习领域开发的，使用直接转换技术的模型转换部署工具也逐年增多，英伟达官方出品的 torch2trt 就是一个典型的类型例子：

torch2trt 支持用户编写转换代码，直接从 PyTorch 转换到，利用 TensorRT Python API，代码开源，链接在此。此转换器的工作原理是将转换函数附加到原始 PyTorch 函数调用，输入数据通过网络传递，每当遇到注册函数就会调用相应的转换器。转换器传递原始 PyTorch 函数的参数和返回语句，以及正在构建的 TensorRT 网络。原始 PyTorch 函数的输入张量被修改为具有一个属性，转换函数使用它向 TensorRT 网络添加层，然后设置相关输出张量的属性。一旦模型完全执行，最终的张量返回被标记为 TensorRT 网络的输出。

这种端到端的模型转换，与选择一个中间格式的中心化转换方法相比，流程相对简单，往往是整个平台完全自研自主使用的，方便部署且具有拓展性；其缺点是灵活性不够，两端的框架需完全确定对应。

2. 中间键转换技术

中间键转换技术的实质是：利用以 ONNX、Caffe 等为代表的开放式模型保存文件规范，从而使得不同深度学习框架下的模型可以采用相同的格式存储数据并交互。[12] 这种技术自 ONNX 出现以来，发展迅速，目前已经形成了强大的深度学习联盟，一般流行的模型部署流水线是：

1. 在任意一种深度学习框架下定义网络结构，训练得到网络参数
2. 将模型的结构和参数转换成一种只描述网络结构的中间表示。针对网络结构的优化会在这步进行。ONNX、Caffe 都是模型的中间表示类型。
3. 用面向硬件的高性能编程框架 (如 CUDA，OpenCL）编写能高效执行深度学习网络中算子的推理引擎，推理引擎将中间表示转换成特定的文件格式，并在对应硬件平台上高效运行。常见的推理引擎有 ONNX Runtime，TensorRT，ncnn ，openppl，OpenVINO 等等。

ONNX(Open Neural Network Exchange) 是一个开放的生态社区，由微软、亚马逊、Facebook 和 IBM 等公司共同开发和维护，代码开源，链接在此。它提供了一种新的深度学习模型和传统 ML 模型的存储格式，称之为 ONNX 格式，而由这种格式定义的模型被称为 ONNX 模型，生成 ONNX 模型的工具被称为 ONNX 工具，该工具的主要开发使用语言是 python。ONNX 可以让开发人员根据项目的发展需求选择合适的开发框架，减少了项目开发的难度并提升了开发效率。

Caffe 提供了用于模型部署和推理的轻量级的 Caffe 模型文件，作为中间表示类型使用。其中 Caffe 使用自己定义的模型文件（.prototxt）和权重文件（.caffemodel）来定义和存储深度学习模型。这个中间转换格式可以很容易地被 Caffe 本身、各种 Caffe 后端库和工具所理解和使用，它还能通过第三方工具或库进行转换，以便在其他框架中进行使用。代码开源，链接在此。

ONNX 和 Caffe 都可以做中间表示类型进行模型转换部署，两者对比如下：

	ONNX	Caffe
灵活性	高	低
op 粒度	细粒度	粗粒度
条件分支	不支持	支持
动态 shape	支持	支持

表 1: ONNX 和 Caffe 对比结果

目前已经出现了部分深度学习框架如 Pytorch、MxNet 等，与 Caffe、ONNX 模型进行直接加载和推理转换的工具，比较典型的例子包括 PyTorch 自带的将模型转换成 ONNX 的 torch.onnx.export 函数：

1. torch.onnx.export [11]

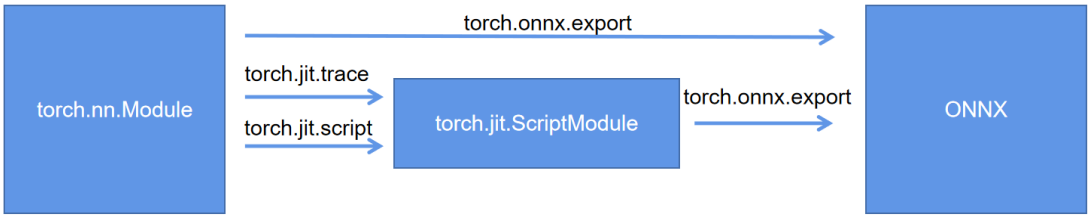

图 1: 计算图导出流程图

实际上,torch.onnx.export 函数的实现过程中也运用了中间键转换的思想。一个 torch.nn.Module 模型在被转换成 ONNX 模型时，会首先被转换成 TorchScript 格式的 torch.jit.ScriptModule 模型。TorchScript 是一种序列化和优化 PyTorch 模型的格式，用以解决 Pytorch 难以部署的问题，它包括代码的追踪及解析、中间表示的生成等各种功能。

torch.onnx.export 需要首先将传入的普通 Pytorch 模型 (torch.nn.Module) 转换成一个 TorchScript 模型 (torch.jit.ScriptModule)。这里有两种转化方法，一种是跟踪法，一种是记录法。跟踪法只能通过实际运行一遍模型的方法导出模型的静态图，即无法识别出模型中的控制流；记录法能通过解析模型来正确记录所有的控制流。由于推理引擎对静态图的支持更好，因此在实际模型应用时，往往不需要将 PyTorch 模型转成 TorchScript 模型显式导出，而使用 torch.onnx.export 跟踪导出即可。

由于各框架下的模型保存方式存在较大的区别、深度学习框架种类繁多，因此各转换工具的开发还处于积极完善的阶段。为了便于这些转换工具的管理，部分开发者们还将这写模型转换工具整合起来形成转换器，如微软开发的 MMdnn 模型转换器。[10] 这些模型转化器允许使用人员在模型转换过程中只向转换器传入待转换的模型文件并设置转换方式，转换器便会根据用户的选择自动生成目标框架下的模型文件，非常方便简洁。下面介绍一个小型的转换器 brocolli：

2. brocolli

brocolli 转换器中实现了 torch2caffe 和 torch2onnx，代码开源，链接在此。

在 pytorch_graph.py 文件中，它使用了模型运行追踪的方法来建立模型的静态表达，“BrocolliTracer 类”和“GraphModule 类”来追踪模型的运行过程并构建计算图。在追踪过程中，进行节点的修剪和形状推断，以下是几个重要函数：

1. 在初始化函数中，初始化 PytorchGraph 类的实例，接收模型的类型、输入数据、具体参数和是否动态批处理参数。并根据模型的类型，选择相应的构建方式。

2. 在 replace 函数中替换模型中的特定子模块，并将转换后的模块设置为原模型的属性。

3. 在 placeholder_prune 函数中实现对计算图进行修剪，删除不必要的占位符节点。遍历计算图中的节点，如果当前节点的操作为"placeholder" 且下一个节点的操作为"call_function"，则进一步检查下一个节点的目标函数名称。如果目标函数为"eq" 且下下个节点的操作为"call_function"，则再次检查下下个节点的目标函数名称。如果目标函数为"_assert"，则删除这三个节点。

4. 在 shape_inference 函数中实现形状推断。使用 BrocolliShapeRunner 类进行形状推断，根据输入数据和具体参数（如果提供），运行形状推断过程。

在 pytorch_caffe_parser.py 文件中，它将 PyTorch 模型的层和操作映射到相应的 Caffe 层，即替换为等效的自定义算子，实现了将 PyTorch 模型转换为 Caffe 模型的解析过程。如：卷积层使用 Deconvolution 算子来表示，设置了卷积核大小、步长、填充等参数。ReLU 层使用 ReLU 算子表示，没有额外的参数设置。池化层使用 Pooling 算子表示，设置了池化类型、池化核大小、步长等参数。Flatten 层使用 Flatten 算子表示，没有额外的参数设置等。然后根据这些不同的层类型和参数设置，生成对应的 Caffe 层对象，并将这些层组合成一个 Caffe 模型。实际上 Brocolli 优化了 pytorch 的导出规则，把 scales 作为一个 attribute 传给了 Resize op，这样解析时只需要解析属性字段即可。

需要注意的是，中间转换过程可能会导致一些模型结构、操作或精度的损失。因此，在进行转换之前，建议进行详细的验证和测试，以确保转换后的模型能够正确并且准确地执行推断。

(三) 图像预处理

在本节中，我们调研了一些常见的图像预处理方法及其对模型推理准确度的影响：

1. 尺寸调整：将图像调整为模型的输入尺寸。适当的尺寸调整可以确保图像在模型中得到正确处理，避免失真和信息丢失。过大或过小的调整可能会导致模型性能下降。

2. 标准化：对图像进行标准化处理，使其具有零均值和单位方差。标准化可以提高模型的收敛速度和稳定性，但对于某些任务和模型，不进行标准化也可能获得良好的结果。

3. 裁剪和填充：根据模型的输入要求，对图像进行裁剪或填充。适当的裁剪和填充可以确保模型接收到适当大小的输入，但不当的操作可能会引入噪声或损失重要信息。

4. 色彩空间转换：将图像从一种色彩空间转换为另一种，例如从 RGB 到灰度或 HSV。对于某些任务，特定的色彩空间可能包含更有用的信息，因此转换可能会对模型的准确度产生影响。

5. 数据增强：应用各种变换和扩充技术，如平移、缩放、旋转、剪切等，以生成更多的训练样本。数据增强可以提高模型的泛化能力，使其在不同变体和视角下更具鲁棒性。[14]

6. 图像滤波：在图像形成和传输过程中，由于设备的不完善性及物理限制，所获得的图像往往达不到期望的质量，因此原始图像的去噪与恢复是非常必要的，可以使用线性滤波技术、非线性滤波技术或者两者结合的技术实现图像滤波，比如人们基于小波变换的低熵性、多分辨性、去相关性等特点将其应用到图像滤波中，可以达到改善修复原图、提高模型准确度的效果。[13]

综上所述，适当的图像预处理方法可以改善模型的推理准确度，提高模型的鲁棒性和泛化能力。我们应根据具体任务和模型的特点对预处理方法进行调整和优化，综合考虑多个因素来确定最佳的预处理策略。

二、 模型介绍

(一) 原理说明

本次适用的模型为 torchvision 内的预训练模型 resnet18。源码地址

随着网络深度的增加，精度变得饱和，然后迅速退化，这是因为网络可能收敛到局部最优，而非全局最优。resnet 的提出主要是为了解决训练网络深度加深但准确率并没有提高，甚至不如浅层网络的问题。

这些问题主要由两个原因产生:

1. 首先，一个深层网络存在梯度消失或梯度爆炸的问题。在反向传播过程中，用于调整网络权重（包括卷积核的值、隐藏层的权重和偏置）的梯度可能会出现极端情况的缩减或增大。反向传播是通过链式法则计算每一层网络权重的梯度，这导致梯度值会进行一系列的连乘。如果每一层的梯度是小于 1 的数，那么在反向传播的过程中，每向前传播一次，都要乘以一个小于 1 的梯度值。随着网络的深度增加，乘以小于 1 的梯度值的次数也增加，从而导致梯度趋近于零。相反，如果每一层的梯度是大于 1 的数，就会发生梯度爆炸的情况。这种梯度消失或梯度爆炸现象阻碍了网络的收敛。然而，通过使用标准初始化方法或在网络的中间层进行归一化，可以解决梯度消失或梯度爆炸的问题。

2. 第二个原因就是如果解决了梯度消失的问题后，仍然会存在层数深的网络没有浅的网络效果好，这就是退化问题。当我们尝试在一个已经学习到的较浅模型的基础上添加额外的层时，这些附加的层被设置为恒等映射（identity mapping）。然而，在实际情况中，这些附加层很难被训练成完全恒等的映射，从而导致了浅层和深层之间的误差差异。

(二) 残差块

为了解决第二个问题，论文作者提出了残差结构。在 ResNet 论文中，有两种映射方式：恒等映射（identity mapping）和残差映射（residual mapping）。恒等映射指的是通过右侧标有”x”的曲线来表示，而残差映射则指的是表示为 $F(x)$ 部分。最终的输出是 $F(x)$ 加上输入 x，即 $F(x)+x$。

ResNet 的优点在于，如果一个网络已经达到了最优状态，继续加深网络的层数可能会导致其他网络出现退化问题（错误率上升）。然而，对于 ResNet 网络来说，当网络已经接近最优状态时，残差映射中的 F(x) 部分会逐渐趋近于零，只剩下恒等映射部分。这意味着理论上网络将一直处于最优状态，网络的性能不会随着深度的增加而降低。这种残差映射的设计使得 ResNet 能够更好地训练非常深的神经网络，并获得更好的性能。

残差块的图示如图2所示。

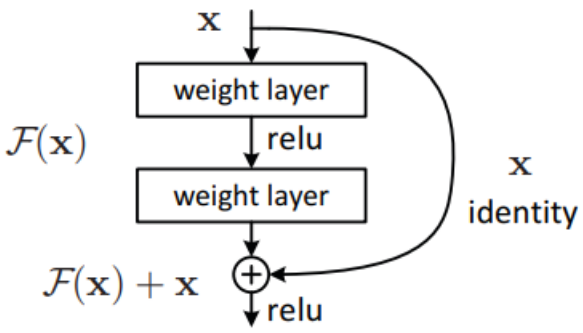

图 2: Residual learning: a building block

残差块中的路径 x 为 identity mapping 恒等映射，也可称之为 shortcut。shortcut 路径大致可以分成 2 种，取决于残差路径是否改变了 feature map 数量和尺寸。

- 一种是将输入 x 原封不动地输出。
- 另一种是经过 1×1 的卷积来升维或者降采样，主要作用是将输出与 $F(x)$ 路径的输出保持 shape 一致，对网络性能的提升并不明显。

两种 shortcut 结构如图3所示。

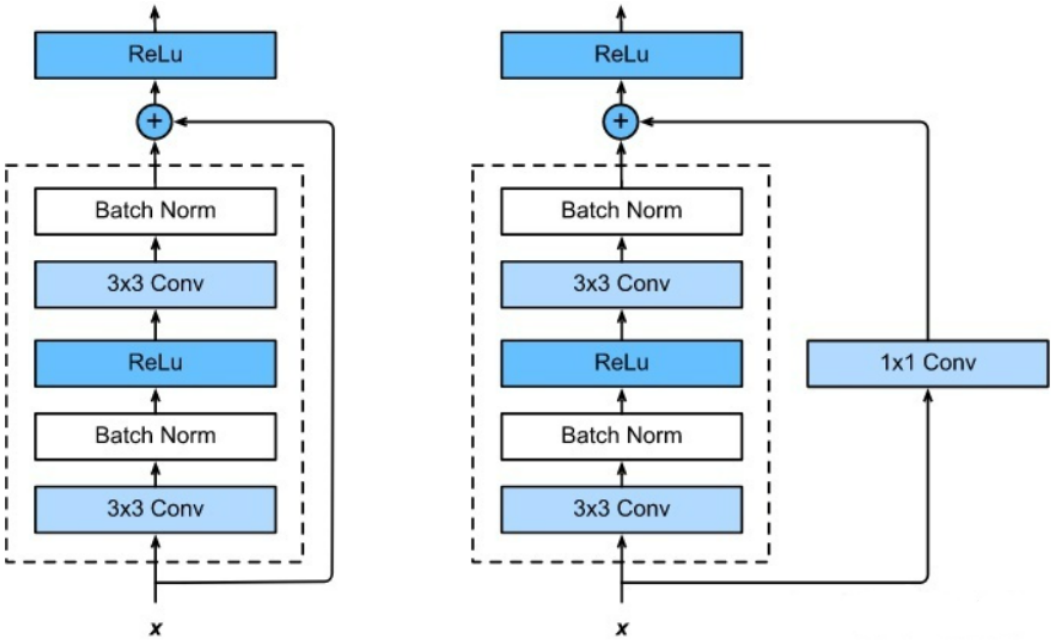

图 3: shortcut structure

作者使用了两种残差块：**Basicblock** 和 **Bottleneck**

BasicBlock 应用在较浅的网络中，如：Resnet 18 和 Resnet 34；而 **Bottleneck** 应用在较深的网络中，如：Resnet 50 Resnet 101 Resnet 152

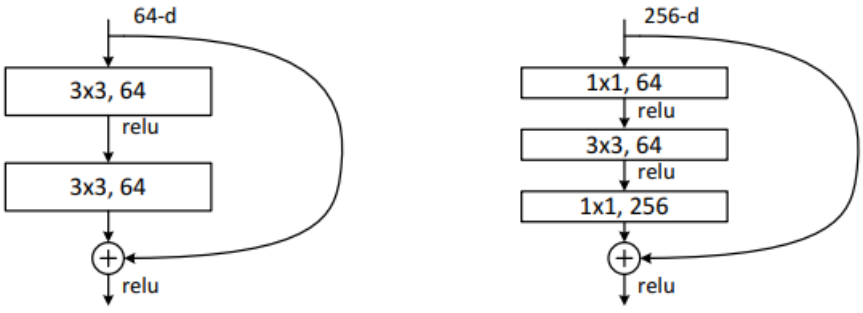

图 4: residual structure

1. BasicBlock

- 输入输出通道数均为 64，有两个 3×3 的卷积层
- 卷积层中参数数量为：$64 \times 64 \times 3 \times 3 + 64 \times 64 \times 3 \times 3 = 73728$

2. Bottleneck

- 两个 1×1 的卷积层，1 个 3×3 的卷积。1×1 卷积层的优势是在更深的网络中，用较小的参数量处理通道数很大的输入。在这个网络中，两个 1*1 的卷积分别负责减少通道数量和恢复通道数量：如输入通道数为 256，1×1 卷积层会将通道数先降为 64，经过 3×3 卷积层后，再将通道数升为 256。
- 整体参数量为： $1\times1\times256\times64+3\times3\times64\times64+1\times1\times64\times256=69632$

如果不采用 Bottleneck 结构，而是使用两个 3×3 的卷积，参数数量为：

$$3\times3\times256\times256+3\times3\times256\times256=1179648$$

因此 bottleneck 结构可以减少参数数量，方便训练，减少算力消耗。

(三) 网络结构

由于我们使用的是 resnet18，因此重点解释 resnet18 的结构。

ResNet18 的 18 层代表的是带有权重的 18 层，包括卷积层和全连接层，不包括池化层和 BN 层。$18=1+2*2*4+1$，第一个 1 位 7×7 的卷积层，最后一个为 FC 层，每个 conv n_x 有 4 个卷积层。

1. layer 1

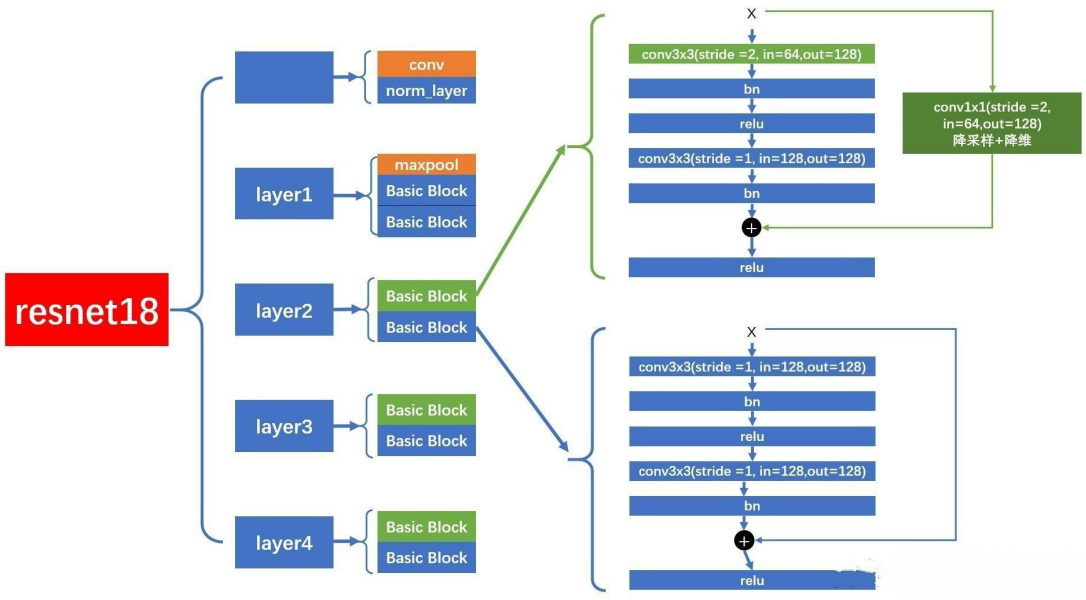

图 5: layer 1

在 layer1 中的每个 BasicBlock 中，特点是没有进行降采样。这意味着卷积层的 stride 设置为 1，不会对特征图的尺寸进行降采样。在进行 shortcut 连接时，也没有经过 downsample 层。

这种设计选择使得 layer1 中的每个 BasicBlock 的输入特征图和输出特征图的尺寸保持一致，没有发生尺寸的减小。这样的设计可以帮助保留更多的空间信息，并减少信息的丢失。此外，由于没有经过 downsample 层，也避免了降采样引入的计算量增加。

2. layer 2/3/4

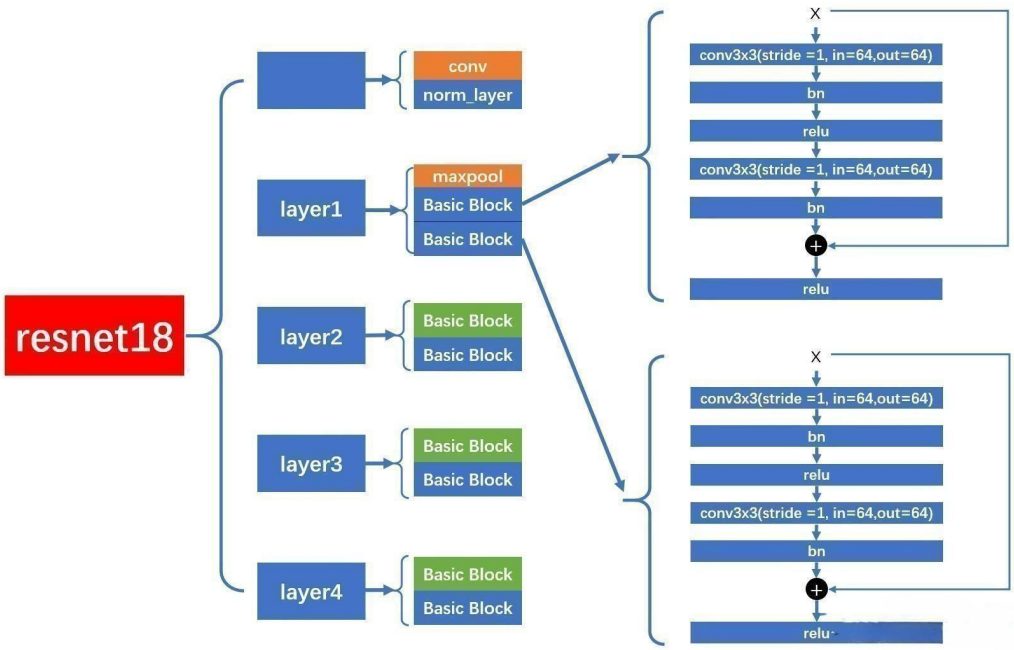

图 6: layer 2/3/4

在剩下的三个 layer 中，每个 layer 包含两个 BasicBlock，第一个 BasicBlock 的第一个卷积层的 stride 被设置为 2，会进行下采样。

在进行 shortcut 连接时，这些 layer2、layer3 和 layer4 的 BasicBlock 需要经过 downsample 层。downsample 层的作用是执行降采样和降维的操作，以匹配降采样后的特征图与残差连接的维度。这样可以确保残差连接的维度一致，从而实现有效的信息传递和梯度流动。

这种设计选择在 layer2、layer3 和 layer4 中引入了降采样操作，有助于增加网络的感受野和表示能力。通过在这些层中引入降采样和降维，ResNet18 可以更好地处理较大规模的任务，捕捉更广泛的上下文信息，并提供更强大的表达能力。

3. 总体架构

resnet18 的总体架构如图7所示:

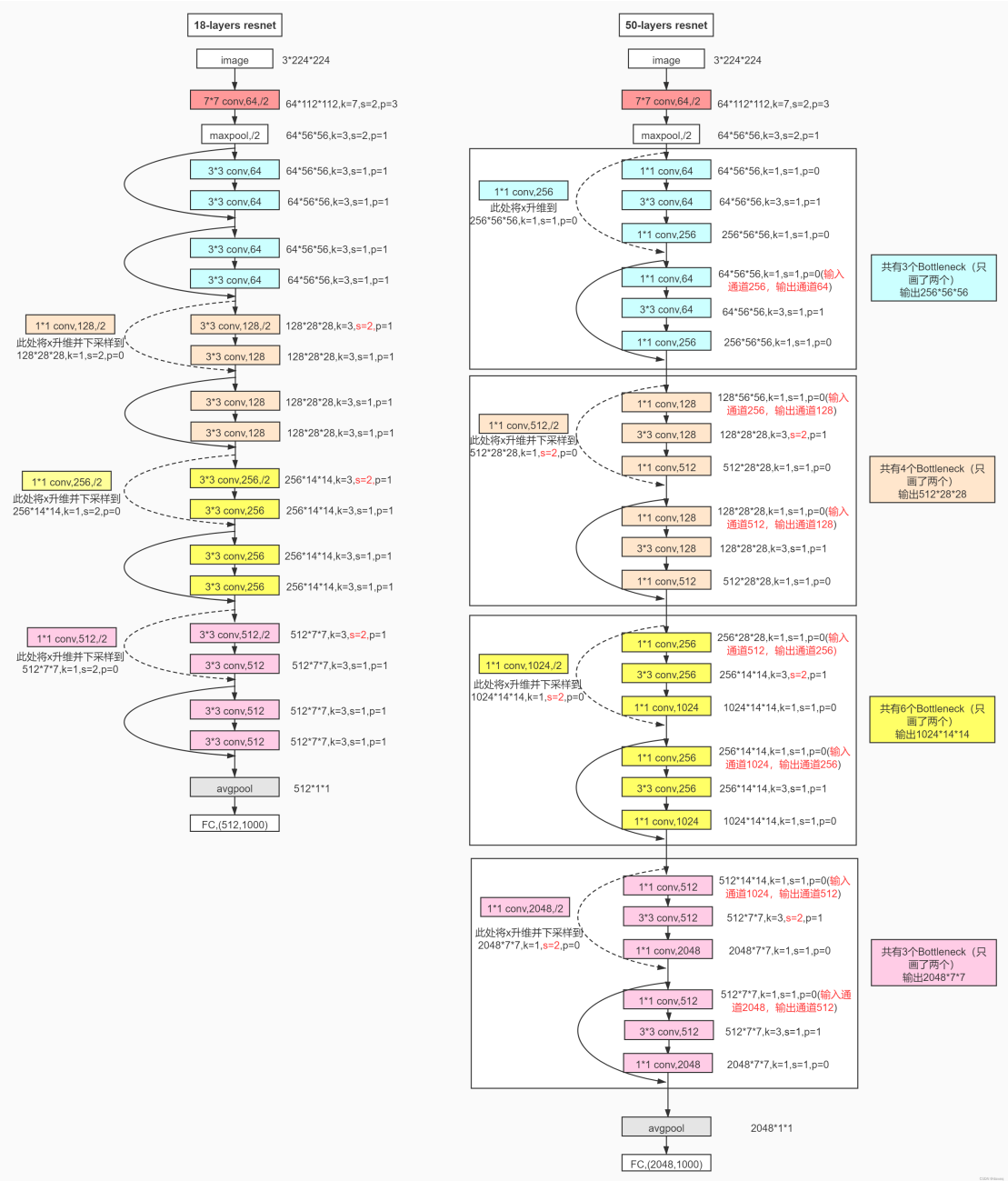

图 7: resnet 18 architecture

（四） 源码解读

接下来，我们将重点解释 resnet 中的重要的类与函数。

1. BasicBlock

BasicBlock

```
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
```

```
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError("BasicBlock only supports groups=1 and
                base_width=64")
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in
                BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input
            when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

BasicBlock 类是 ResNet 中的基本模块之一，它定义了 ResNet 中的基本残差块，通过堆叠多个 BasicBlock 可以构建更深的 ResNet 网络，实现更好的图像分类、目标检测等计算机视觉任务。主要功能如下：

初始化函数（___init___）：

- 接收输入参数，包括输入通道数（inplanes）、输出通道数（planes）、步长（stride）、下采样模块（downsample）等。

- 初始化 BasicBlock 的各个层，包括两个卷积层（self.conv1 和 self.conv2）、批归一化层（self.bn1 和 self.bn2）、ReLU 激活函数（self.relu）等。
- 设置其他属性，如步长（self.stride）。

前向传播函数（forward）：

- 接收输入张量 x 作为输入。
- 通过卷积、批归一化和 ReLU 激活函数依次处理输入。
- 如果存在下采样模块（self.downsample），对输入进行下采样得到 identity，以便在后面进行残差连接。在默认情况 downsample=None，表示不做 downsample，但有一个情况需要做，就是一个 BasicBlock 的 x 要与 F(x) 相加时，若 x 和 F(x) 的通道数不一样，则要做一个 downsample。方法：用一个 1×1 的卷积核处理 x，变成想要的通道数。
- 将处理后的特征图与 identity 相加，实现残差连接。
- 最后再通过 ReLU 激活函数得到最终输出特征图 out。
- 返回最终输出特征图 out。

2. Bottleneck

Bottleneck 类似于 BasicBlock，但是具有更深的网络结构，用于构建 resnet 中的瓶颈块。由于 resnet18 中并不涉及 Bottleneck，但是 Bottleneck 层是一个神经网络中相当重要的结构，故这里仍作简要介绍。

主要功能如下：

初始化函数（___init___）：

- 接收输入参数，包括输入通道数（inplanes）、输出通道数（planes）、步长（stride）、下采样模块（downsample）等。
- 初始化 Bottleneck 的各个层，包括三个卷积层（self.conv1、self.conv2 和 self.conv3）、批归一化层（self.bn1、self.bn2 和 self.bn3）、ReLU 激活函数（self.relu）等。
- 设置其他属性，如步长（self.stride）。

前向传播函数（forward）：与 BasicBlock 大致相同，但是卷积层为 $1\times1->3\times3->1\times1$。

Bottleneck 类在 ResNet 中引入了瓶颈结构，通过 1x1 卷积层（self.conv1 和 self.conv3）来降低维度和增加非线性性，并在中间使用 3x3 卷积层（self.conv2）进行特征提取。这种设计可以有效地减少参数数量，提高计算效率，并在一定程度上改善模型的表达能力。

通过堆叠多个 Bottleneck 模块，可以构建更深层次的 ResNet 网络，用于处理更复杂的计算机视觉任务，如图像分类、目标检测和语义分割等。

3. 网络主体

resnet 共有 5 个阶段。

第一阶段为一个 7*7 的卷积，stride = 2，padding = 3，然后经过 BN、ReLU 和 maxpooling（3×3 maxpooling stride=2），此时特征图的尺寸已成为输入的 1/4。

stage 1

```
self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
    bias=False)
self.bn1 = norm_layer(self.inplanes)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```

第二/三/四/五阶段：也就是代码中 layer1，layer2，layer3，layer4。这里用 _make_layer 函数产生四个 Layer，需要用户输入每个 layer 的 block 数目（即 layers 列表）以及采用的 block 类型。

stage 2/3/4/5

```
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2, dilate=
    replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2, dilate=
    replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2, dilate=
    replace_stride_with_dilation[2])
```

其中，make_layer 函数的参数如下：

- **block**：选择要使用的模块是 BasicBlock 还是 Bottleneck 类。由于使用的是 resnet18，我们应该选择 BasicBlock 类。
- **planes**：当前块的基准通道数（每个 layer 中间层的通道数），64->128->256->512，而每个 block 的输出通道数 = 基准通道数 × expansion（basicblock 的 expansion=1; bottleneck 的 expansion=4）
- layers：4 个 blocks 中分别包含多少个残差结构，如：resnet18：[2, 2, 2, 2]

平均池化 + 全连接：

avg_pool+fc

```
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

输入神经元个数 = 512 * block.expansion。在这二者之间还有一个 flatten 层，这将会成为任务二的关键所在。

4. 激活函数与损失函数

在 ResNet 的训练过程中，通常采用的激活函数是 ReLU（Rectified Linear Unit）激活函数。ReLU 激活函数定义为：f(x) = max(0, x)，其中 x 是输入。ReLU 函数将负值设为零，保持正值不变，具有线性和非线性特性，被广泛应用于深度学习中。

至于损失函数，根据具体的任务类型而有所不同。对于分类任务，常用的损失函数是交叉熵损失函数（Cross-Entropy Loss）。交叉熵损失函数是衡量实际输出与期望输出之间差异的指标，它在分类问题中表现良好。对于回归任务，常用的损失函数可以是均方误差（Mean Squared Error）或平均绝对误差（Mean Absolute Error），具体选择取决于问题的特点。

三、 实验分析

(一) 任务一

1. 实现逻辑

在任务一中，本小组选择 Resnet18 图像分类网络，实现了将 Pytorch 模型分别转换成 Caffe 模型与 ONNX 模型，并针对同一张测试图片，测试打印了三种模型下的推理结果置信度以及对于同一测试图片进行图像推理的模型吞吐量。置信度结果相差不大，可以忽略 Pytorch 向 Caffe 模型、ONNX 模型转换后的精度损失。具体实现流程可以概括成为两部分：

1. 一部分是模型转换，根据需要导入头文件包，加载 pytorch 中的图像分类网络，然后使用工具链，分别将 pytorch 模型转换成 caffe 模型和 onnx 模型；
2. 第二部分是模型推理，这其中又分为两部分，首先需要加载图片并进行预处理、加载标签文件；然后分别运行 pytorch 模型、caffe 模型和 onnx 模型对同一图片的推理结果，从置信度和吞吐量两方面进行对比。

核心代码主要在于模型转换和模型推理这两阶段，下面分别介绍 pytorch2caffe 和 pytorch2onnx 的核心代码：

1. pytorch2caffe

1. 模型转换：使用 brocolli 转换器中提供的转换工具，调用 PytorchCaffeParser 函数按照输入的参数将 pytorch 模型转换成 caffe 模型，该转换器转换的实现逻辑在背景调研部分已经详细介绍，此处不赘述，存储转换后的 caffe 模型：

```
x = torch.rand(1, 3, 224, 224)
pytorch_parser = PytorchCaffeParser(net, x)
pytorch_parser.convert()
pytorch_parser.save('resnet18')
```

2. 模型推理：首先加载刚刚转换成功的模型文件和权重文件，然后定义输入数据的形状。此时，经过预处理后的图片输入为 tensor，将其输入到网络中。进行前向推理，反复测试 1000 次，从而获得输出并打印打印置信度 top5 和对应的标签和对应的吞吐量

```
model_file = './resnet18.prototxt'
weight_file = './resnet18.caffemodel'
net = caffe.Net(model_file, weight_file, caffe.TEST)
net.blobs['x'].reshape(1, 3, 224, 224)
net.blobs['x'].data[...] = tensor

NUM_IMAGES = 1000  # 测试推理的图像数量
start_time = time.time()
for _ in range(NUM_IMAGES):
    net.forward()# 进行前向推理
    output = net.blobs['fc'].data[0] #获取网络的输出
end_time = time.time()

#计算打印置信度top5和对应的标签
softmax = np.exp(output) / np.sum(np.exp(output))
```

```
sorted_indices = np.argsort(-softmax)
top_5_indices = sorted_indices[:5]
for idx in top_5_indices:
    print(categories[idx], softmax[idx])

#计算打印吞吐量
total_time = end_time - start_time
throughput = NUM_IMAGES / total_time
print(f'Throughput: {throughput} images/second')
```

2. pytorch2onnx

1. 模型转换：使用 PyTorch 自带的 torch.onnx.export 函数按照输入的参数将 pytorch 模型转换成 ONNX 模型，该转换器转换的实现逻辑在背景调研部分已经详细介绍，此处不赘述，存储转换后的 onnx 模型：

```
net = models.resnet18(pretrained=True)
x = torch.rand(1, 3, 224, 224)
torch.onnx.export(net, x, 'resnet18.onnx')
```

2. 模型推理：使用 onnxruntime 推理引擎加载运行刚刚转换成功 onnx 模型。此时，经过预处理后的图片输入为 tensor，将其输入到网络中。进行前向推理，反复测试 1000 次，从而获得输出并打印打印置信度 top5 和对应的标签和对应的吞吐量

```
model = ort.InferenceSession("resnet18.onnx")
input_name = model.get_inputs()[0].name
output_name = model.get_outputs()[0].name

NUM_IMAGES = 1000  # 测试推理的图像数量
start_time = time.time()
for _ in range(NUM_IMAGES):
    outputs = model.run([output_name], {input_name: tensor.numpy()})
    output_tensor = torch.Tensor(outputs[0])
end_time = time.time()

#计算打印置信度top5和对应的标签
probabilities = torch.nn.functional.softmax(output_tensor[0], dim=0)
top5_prob, top5_catid = torch.topk(probabilities, 5)
for i in range(top5_prob.size(0)):
    print(categories[top5_catid[i]], top5_prob[i].item())

#计算打印吞吐量
total_time = end_time - start_time
throughput = NUM_IMAGES / total_time
print(f'Throughput: {throughput} images/second')
```

2. 实现结果

1. 置信度分别打印比较 pytorch、caffe、ONNX 三种模型下的推理结果置信度 top5，结果如下表所示：

label	Samoyed	white wolf	Arctic fox	Pomeranian	West Highland white
Pytorch	0.89802831	0.04398656	0.03794151	0.00478651	0.00336607
Caffe	0.89802784	0.0439865	0.03794145	0.00478651	0.00336607
ONNX	0.89802789	0.04398662	0.03794185	0.00478651	0.00336608

表 2: pytorch、caffe、ONNX 三种模型置信度对比结果

结合上表结果，我们发现针对同一张测试图片，测 pytorch、caffe、ONNX 三种模型的推理结果置信度结果相差不大，可以忽略 Pytorch 向 Caffe 模型、ONNX 模型转换后的精度损失，转换成功！

2. 吞吐量分别打印比较 pytorch、caffe、ONNX 三种模型传入图片加载部署 1000 次每次平均的吞吐量，结果如下表所示，单位是 images/second：

	Pytorch		Caffe	ONNX	
吞吐量	96.67	16.41	93.26	0.00478651	0.00336607

表 3: pytorch、caffe、ONNX 三种模型吞吐量对比结果

对于每种模型都检测其获得图片输入后到加载部署完成的时间，可以发现由于 caffe 模型需要加载模型文件和权重文件，并要根据权重文件进行前向传播再获得输出结果，因此其吞吐量明显更低速度更慢，Pytorch 模型和 ONNX 模型的结果相差接近。

(二) 任务二

任务二要求实现 Caffe2Darknet 的模型转换，并解决遇到的一系列不能正常转换模型、框架间算子不兼容、不同框架下算子实现的功能一致但名称不一致等问题。

在代码中，已经实现了 caffe 至 darknet 的转换函数，但是需要解决 Flatten 在 DarkNet 上的实现。直觉是，我们需要在 Darknet 中添加一个 Flatten 算子，重新编译 DarkNet；然后，修改转换函数，以支持 Flatten 层的实现。然而事情并非如此，通过观察维度的变化，我们发现 Flatten 层被融入到了 fc 层中。如果直接去掉 Flatten，模型也可以运行，但是分类结果不正确(思路二)。我们将尽力完成这一项工作，下面我们将阐述我们的思路。

1. 思路一：改进模型转换工具

我们需要改进 **caffe2darknet**.py 这个文件。首先阅读源码，理清各个函数的作用：

- _save_weights：将权重数据数组 data 保存到指定的文件 weightfile 中
- _save_cfg：将神经网络配置块保存到文件中
- _print_cfg：打印 Darknet 框架中的神经网络配置块

- _print_cfg_nicely：打印 layer, filters, size, input, output 等信息，并存储 out_widths, out_heights, out_filters 等信息
- _parse_caffemodel：解析 caffe 文件
- _parse_prototxt：解析 Caffe 模型的配置文件，并将其转化为一个有序字典的形式返回，其中包含模型的属性信息（props）和层信息（layers）
- _generate_hash：计算给定权重文件和配置文件的 MD5 哈希值，并返回哈希值的字符串表示
- _check_hash：检查文件的哈希值是否与给定的哈希值相匹配
- class Caffe2Darknet：用于模型转换的类，在其中包含了以上的操作，使用这个类将能完成我们所需要的模型转换。在 convert() 函数中描述了转换的具体过程。

在模型转换的过程中，并没有顾及 Flatten 操作。我们需要在转换时添加对 Flatten 层的描述。

在 Keras 中文文档中有对 Flatten 的解释：Flatten 层用来将输入“压平”，即把多维的输入一维化，常用在从卷积层到全连接层的过渡。Flatten 不影响 batch 的大小。也就是说，Flatten 并没有进行实际的运算，只是在 reshape 数据。

我们首先更改 _print_cfg_nicely，使其能正常打印 Flatten 层：

_print_cfg_nicely

```
def _print_cfg_nicely(blocks):
    print('layer     filters    size              input                output
        ')
    """
    prev_width: 上一层网络的输出宽度
    prev_height: 上一层网络的输出高度
    prev_filters: 上一层网络的通道数
    """
    ...
    for block in blocks:
        ind = ind + 1
        ...
        #展开后的向量长度为 prev_width、prev_height 和 prev_filters相乘。
        elif block['type'] == 'flatten':
            #prev_width = int(block['width'])
            #prev_height = int(block['height'])
            #prev_filters = int(block['filters'])
            flat_size = prev_width * prev_height * prev_filters
            print('%5d %-6s %10d' % (ind, 'flat', flat_size))
            prev_width = flat_size
            prev_height = 1
            prev_filters = 1
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        ...
```

在遍历 block 的时候, 首先识别出 flatten 层, 由于 flatten 是将多维输出一维化, 因此 flat_size = prev_width * prev_height * prev_filters, 然后再重新设置 prev_width = flat_size, prev_height 和 prev_filters 为 1 即可。

然后需要处理的是 Caffe2Darknet 的 convert 函数，在这个函数中实现了模型的转化。convert(self) 方法根据给定的 net 和 weight 文件路径解析 Caffe 模型, 并根据解析结果生成 Darknet 模型的配置信息。这个函数的主要处理逻辑如下:

1. 首先，根据 net 文件解析网络结构信息，获取网络的属性。
2. 然后，根据 weight 文件解析权重信息，遍历每一层，并根据层的类型生成对应的 Darknet 模型配置信息。在这个过程中，还会处理一些特殊层类型，如卷积层、池化层、Eltwise 层和全连接层等。
3. 最后，将生成的 Darknet 模型的配置信息保存到.cfg 文件中，并将权重数据保存到.weights 文件中。同时，打印模型的配置信息，并计算并打印 Darknet 模型的哈希值。

我们需要做的便是在遍历层中增加对 Flatten 的处理。

convert

```
def _print_cfg_nicely(blocks):
def convert(self):
        protofile = self.net
        caffemodel = self.weight
        model = _parse_caffemodel(caffemodel)
        layers = model.layer
        ...
        while i < layer_num:
            layer = layers[i]
            print(i, layer['name'], layer['type'])
            ...
            elif layer['type'] == 'Flatten':
                assert (layer_id[layer['bottom']] == len(blocks) - 1)
                block = OrderedDict()
                block['type'] = 'connected'
                block['output'] = 512
                block['activation'] = 'linear'
                top = layer['top']
                layer_id[top] = len(blocks)
                blocks.append(block)
                i = i + 1
            ...
        print('done')
        assert self.out_file is not None
        weightfile = self.out_file + '.weights'
        cfgfile = self.out_file + '.cfg'
        _save_weights(np.array(wdata), weightfile)
        _save_cfg(blocks, cfgfile)
        _print_cfg(blocks)
        _print_cfg_nicely(blocks)
```

```
        print('Hash of Darknet model has been published: ',
              format(_generate_hash(weightfile, cfgfile)))
```

在模型转换过程中，根据其他类型的层（例如 Convolution、Pooling、Eltwise 等）仿写，Flatten 层被解析并添加到了 blocks 列表中。blocks 列表最终用于构建 Darknet 模型。为了添加 Flatten 层，首先使用断言语句确保该层的输入连接到了 blocks 列表的最后一个块。然后创建一个有序字典 block，用于描述 Darknet 模型的一个块。

考虑到 Darknet 中没有专门的块类型表示 Flatten 操作，而且它被融入到了 fc 层中，因此我们将 block 的类型设置为'connected'，表示该层是一个全连接层，并设置输出节点数为 512，激活函数为线性激活函数。将该层的输出命名为 top，并将其对应的 ID 添加到 layer_id 字典中。最后，将该 block 添加到 blocks 列表中。由于该层已处理完，计数器 i 需自增。

需要注意的是，在实际编写代码时，我们应该将 Flatten 层转换成全连接层的形式，而不是直接使用 block['type'] = 'flatten'。这是因为在 Darknet 中，Flatten 层被表示为具有特定输出节点数的全连接层，目的是将输入数据展平为一维向量，以便后续层可以处理。

2. 思路一：实验结果

在转换后，cfg 文件的最后 4 层如下所示：

resnet18.cfg

```
[shortcut]
from=-3
activation=relu

[avgpool]

[connected]
output=512
activation=linear

[connected]
output=1000
activation=linear
```

可以看到，我们已经把 flatten 层变成了 connected 层。

打印出的网络结构图如图8所示：

```
   layer   filters  size/strd(dil)      input                output
   0 conv     64       7 x 7/ 2    224 x 224 x   3 ->  112 x 112 x  64 0.236 BF
   1 max                3x 3/ 2    112 x 112 x  64 ->   56 x  56 x  64 0.002 BF
Unused field: 'pad = 1'
   2 conv     64       3 x 3/ 1     56 x  56 x  64 ->   56 x  56 x  64 0.231 BF
   3 conv     64       3 x 3/ 1     56 x  56 x  64 ->   56 x  56 x  64 0.231 BF
   4 Shortcut Layer: 1,  wt = 0, wn = 0, outputs:  56 x  56 x  64 0.000 BF
   5 conv     64       3 x 3/ 1     56 x  56 x  64 ->   56 x  56 x  64 0.231 BF
   6 conv     64       3 x 3/ 1     56 x  56 x  64 ->   56 x  56 x  64 0.231 BF
   7 Shortcut Layer: 4,  wt = 0, wn = 0, outputs:  56 x  56 x  64 0.000 BF
   8 conv    128       3 x 3/ 2     56 x  56 x  64 ->   28 x  28 x 128 0.116 BF
   9 conv    128       3 x 3/ 1     28 x  28 x 128 ->   28 x  28 x 128 0.231 BF
  10 route  7                                 ->   56 x  56 x  64
  11 conv    128       1 x 1/ 2     56 x  56 x  64 ->   28 x  28 x 128 0.013 BF
  12 Shortcut Layer: 9,  wt = 0, wn = 0, outputs:  28 x  28 x 128 0.000 BF
  13 conv    128       3 x 3/ 1     28 x  28 x 128 ->   28 x  28 x 128 0.231 BF
  14 conv    128       3 x 3/ 1     28 x  28 x 128 ->   28 x  28 x 128 0.231 BF
  15 Shortcut Layer: 12,  wt = 0, wn = 0, outputs:  28 x  28 x 128 0.000 BF
  16 conv    256       3 x 3/ 2     28 x  28 x 128 ->   14 x  14 x 256 0.116 BF
  17 conv    256       3 x 3/ 1     14 x  14 x 256 ->   14 x  14 x 256 0.231 BF
  18 route  15                                    ->   28 x  28 x 128
  19 conv    256       1 x 1/ 2     28 x  28 x 128 ->   14 x  14 x 256 0.013 BF
  20 Shortcut Layer: 17,  wt = 0, wn = 0, outputs:  14 x  14 x 256 0.000 BF
  21 conv    256       3 x 3/ 1     14 x  14 x 256 ->   14 x  14 x 256 0.231 BF
  22 conv    256       3 x 3/ 1     14 x  14 x 256 ->   14 x  14 x 256 0.231 BF
  23 Shortcut Layer: 20,  wt = 0, wn = 0, outputs:  14 x  14 x 256 0.000 BF
  24 conv    512       3 x 3/ 2     14 x  14 x 256 ->    7 x   7 x 512 0.116 BF
  25 conv    512       3 x 3/ 1      7 x   7 x 512 ->    7 x   7 x 512 0.231 BF
  26 route  23                                    ->   14 x  14 x 256
  27 conv    512       1 x 1/ 2     14 x  14 x 256 ->    7 x   7 x 512 0.013 BF
  28 Shortcut Layer: 25,  wt = 0, wn = 0, outputs:   7 x   7 x 512 0.000 BF
  29 conv    512       3 x 3/ 1      7 x   7 x 512 ->    7 x   7 x 512 0.231 BF
  30 conv    512       3 x 3/ 1      7 x   7 x 512 ->    7 x   7 x 512 0.231 BF
  31 Shortcut Layer: 28,  wt = 0, wn = 0, outputs:   7 x   7 x 512 0.000 BF
  32 avg                           7 x   7 x 512 ->    512
  33 connected                           512  ->   512
  34 connected                           512  ->  1000
Total BFLOPS 3.630
avg outputs = 114812
```

图 8: caffe2darknet architecture1

可以看到, 倒数第二层和倒数第一层都为 connected 层, 维度变化为 512->512 和 512->1000. 推理 top5 结果如图9所示:

```
[('cocker spaniel', 5.662687301635742),
 ("yellow lady's slipper", 5.231784820556641),
 ('umbrella', 4.674552917480469),
 ('Brittany spaniel', 4.525413513183594),
 ('balloon', 4.159382343292236)]
```

图 9: caffe2darknet outcome1

虽然能够进行推理，但很明显模型的输出是错误的，甚至置信度 >1，非正确的结果。

3. 思路二：直接去掉 flatten

我们的思路为，既然 darknet 的 fc 中已经包括了 flatten 操作，那如果我们直接去掉 flatten 层是否也能运行？于是我们的操作变为：

1. 首先使用 Caffe2Darknet 执行转化操作
2. 修改 resnet18.cfg，去掉 flatten 层
3. 使用修改过后的 cfg 文件进行推理

因此 resnet18.cfg 的最后三层变为：

resnet18.cfg

```
[shortcut]
from=-3
activation=relu

[avgpool]

[connected]
output=1000
activation=linear
```

4. 思路二：实验结果

去掉 flatten 后的网络结构与思路一相同，此处略过，但推理结果有所不同。

```
[('mink', 5.748802661895752),
 ('Sealyham terrier', 5.4668354988098145),
 ('dugong', 5.058732986450195),
 ('cocker spaniel', 4.630618095397949),
 ('bighorn', 4.518701076507568),
```

图 10: caffe2darknet outcome2

从图中可以看出，top5 的结果为 mink, sealyham terrier, dugong, cocker spaniel, bighorn，而且置信度也都大于 1，说明实验并不成功。

5. 思路三：增加 flatten 算子

我们需要处理 **DarkNet**。首先需要注册 FLATTEN 层，在 darknet.h 文件中加入 flatten 类型：

```
85      ISEG,
86      REORG,
87      UPSAMPLE,
88      LOGXENT,
89      L2NORM,
90      BLANK,
91      FLATTEN
92  } LAYER_TYPE;
93
```

图 11: darknet.h 文件

修改 parser.c 文件，以识别 flatten，并返回 FLATTEN 的 LAYER_TYPE:

darknet > src > C parser.c

```
95      if (strcmp(type, "[contrastive]") == 0) return CONTRASTIVE;
96      if (strcmp(type, "[route]")==0) return ROUTE;
97      if (strcmp(type, "[upsample]") == 0) return UPSAMPLE;
98      if (strcmp(type, "[empty]") == 0
99          || strcmp(type, "[silence]") == 0) return EMPTY;
100     if (strcmp(type, "[implicit]") == 0) return IMPLICIT;
101     f (strcmp(type, "[flatten]") == 0) return FLATTEN;
102     return BLANK;
103 }
104
```

图 12: parser.c 文件

仍然在 parser.c 文件中，添加一个 parse_flatten 函数。这个函数将接收参数并返回一个表示层结构的对象，用于初始化并定义层的输出格式。其中，fatten 层的输出尺寸为 h * w * c，其中 w 和 h 均为 1，c 表示经过 fatten 之后的通道数。使用 make_fatten_layer 函数来真正创建这个层。

```
layer parse_flatten(list *options, size_params params)
{
    layer l = make_flatten_layer(params.batch, params.inputs);
    l.out_h = 1;
    l.out_w = 1;
    l.out_c = params.h * params.w * params.c;
    l.h = params.h;
    l.w = params.w;
    l.c = params.c;

    return l;
}
```

图 13: parse flatten

在 parser.c 文件中的 parse_network_cfg_custom 函数中添加一个 FLATTEN 层的识别:

```
    }else if(lt == ACTIVE){
        l = parse_activation(options, params);
    }else if(lt == FLATTEN){
        l = parse_flatten(options, params);
    }else if(lt == LOGXENT){
        l = parse_logistic(options, params);
```

图 14: flatten 层识别

仿照其他层添加 flatten_layer.h:

```
#ifndef FLATTEN_LAYER_H
#define FLATTEN_LAYER_H

#include "layer.h"
#include "network.h"

#ifdef __cplusplus
extern "C" {
#endif
layer make_flatten_layer(int batch, int inputs);

void forward_flatten_layer(layer l, network_state state);
void backward_flatten_layer(layer l, network_state state);

#ifdef GPU
void forward_flatten_layer_gpu(layer l, network_state state);
void backward_flatten_layer_gpu(layer l, network_state state);
#endif

#ifdef __cplusplus
}
#endif
```

图 15: flatten layer.h

仿照其他层添加 flatten_layer.c:

```
layer make_flatten_layer(int batch, int inputs)
{
    layer l = { (LAYER_TYPE)0 };
    l.type = FLATTEN;

    l.inputs = inputs;
    l.outputs = inputs;
    l.batch=batch;

    l.output = (float*)xcalloc(batch * inputs, sizeof(float));
    l.delta = (float*)xcalloc(batch * inputs, sizeof(float));

    l.forward = forward_flatten_layer;
    l.backward = backward_flatten_layer;
#ifdef GPU
    l.forward_gpu = forward_flatten_layer_gpu;
    l.backward_gpu = backward_flatten_layer_gpu;

    l.output_gpu = cuda_make_array(l.output, inputs*batch);
    l.delta_gpu = cuda_make_array(l.delta, inputs*batch);
#endif
    fprintf(stderr, "Flatten Layer: %d inputs\n", inputs);
    return l;
}

void forward_flatten_layer(layer l, network_state state)
{
    copy_cpu(l.outputs*l.batch, state.input, 1, l.output, 1);
}

void backward_flatten_layer(layer l, network_state state)
{
    copy_cpu(l.outputs*l.batch, l.delta, 1, state.delta, 1);
}
```

图 16: flatten layer.c

最后需要在 src/parser.c, src/network.c 中 include ”flatten_layer.h”，并修改 Makefile 文件，链接 flatten_layer 的目标文件，重新编译。

重新修改 caffe2darknet.py 文件，以支持 darknet 层:

caffe2darknet

```
# _print_cfg_nicely
#输出为prev_width、prev_height 和 prev_filters 相乘
        elif block['type'] == 'flatten':
            flat_size = prev_width * prev_height * prev_filters
            print('%5d %-6s %10d' % (ind, 'flatten', flat_size))
            prev_width = 1
            prev_height = 1
```

```
            prev_filters = flat_size
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)

# class Caffe2Darknet
elif layer['type'] == 'Flatten':
                assert (layer_id[layer['bottom']] == len(blocks) - 1)
                block = OrderedDict()
                block['type'] = 'flatten'
                top = layer['top']
                layer_id[top] = len(blocks)
                blocks.append(block)
                i = i + 1
```

6. 使用官方 resnet18

根据官方的指导，我们尝试使用了 pjreddie 自带的 resnet18.cfg，并得到以下结果：

```
  16 conv    256  3 x 3 / 2    28 x  28               x 128   ->    14 x  14 x 256
  17 conv    256  3 x 3 / 1    14 x  14               x 256   ->    14 x  14 x 256
  18 route  15
  19 conv    256  1 x 1 / 2    28 x  28               x 128   ->    14 x  14 x 256
  20 shortcut 17
  21 conv    256  3 x 3 / 1    14 x  14               x 256   ->    14 x  14 x 256
  22 conv    256  3 x 3 / 1    14 x  14               x 256   ->    14 x  14 x 256
  23 shortcut 20
  24 conv    512  3 x 3 / 2    14 x  14               x 256   ->     7 x   7 x 512
  25 conv    512  3 x 3 / 1     7 x   7               x 512   ->     7 x   7 x 512
  26 route  23
  27 conv    512  1 x 1 / 2    14 x  14               x 256   ->     7 x   7 x 512
  28 shortcut 25
  29 conv    512  3 x 3 / 1     7 x   7               x 512   ->     7 x   7 x 512
  30 conv    512  3 x 3 / 1     7 x   7               x 512   ->     7 x   7 x 512
  31 shortcut 28
  32 avg                      7 x   7 x 512   ->      512
  33 conv   1000  1 x 1 / 1     1 x   1               x 512   ->     1 x   1 x1000
```

图 17: 官方 resnet18

```
b'Samoyed', 0.877466082572937
b'Great Pyrenees', 0.0257264394313097
b'Eskimo dog', 0.01820150762796402
b'Pomeranian', 0.017197703942656517
b'keeshond', 0.009782121516764164
```

图 18: 官方 resnet18

可以看到，官方的 resnet18 实现是正确的，虽然 top5 的值与 pytorch 版的 resnet 有所不同，但是分类正确，且置信度为合理的值。

四、 组内分工

代码分工：二人协作完成

报告分工：管昀玫同学负责模型介绍和任务二的实验分析；宋佳蓁同学负责背景调研和任务一的实验分析

五、 总结

本小组的大作业实现总共分为三个阶段，分别是实验前的背景调研、实验任务一以及实验任务二。

在背景调研的阶段，本小组主要调研了目前常用的深度学习框架进而引出模型部署转换的重要性、目前常用的模型部署转换方法包括直接转换技术和中间键转换技术。在直接转换技术部分主要调研了 torch2trt 工具；在中间键转换技术部分首先调研比较了 ONNX 和 Caffe 两种最常用的中间键，接着调研了深度学习框架中自带的推理转换的工具 torch.onnx.export 和转换器 brocolli。最后调研了图像预处理技术对于模型推理准确度可能的影响。

在任务一阶段，主要使用了之前调研的两种工具 torch.onnx.export 函数和转换器 brocolli，分别实现了将图像分类网络 resnet18 从 pytorch 模型转换成 onnx 模型和 caffe 模型，并比较了他们的准确率和吞吐量。发现 Pytorch 向 Caffe 模型、ONNX 模型转换后的精度损失可以忽略，转换成功。

在任务二阶段，主要进行了四种思路的尝试，思路一改进模型转换工具 caffe2darknet.py，思路二是直接去掉 flatten 层，思路三是增加 flatten 算子，思路四是使用官方的 resnet18，分别实现了四种思路，虽然只有使用官方的 resnet18 的测试结果成功，但是每一种尝试都加深了我们对模型部署与转换的理解，激发我们的探索兴趣，促进了我们的进步!

参考文献

[1] 章敏敏, 徐和平, 王晓洁, 等. 谷歌 TensorFlow 机器学习框架及应用 [J]. 微型机与应用, 2017, 36(10): 58-60.

[2] 杨楠. 基于 Caffe 深度学习框架的卷积神经网络研究 [D]. 石家庄: 河北师范大学,2016.

[3] 宗春梅, 张月琴, 石丁. PyTorch 下基于 CNN 的手写数字识别及应用研究 [J]. 计算机与数字工程, 2021, 49(06): 1107-1112.

[4] Seime B C, Mannino M, Runde T. Two Approaches to Particle Simulation: OpenMPI and CUDA[J]. Hgpu Org, 2013.

[5] 董如意, 杜俊杰. 基于 Paddle Paddle 云平台的机器学习课程实验案例研究 [J]. 无线互联科技, 2021, 18(21): 134-135.

[6] Wang Y , Yang Z , Liu T , et al. Passivity and Synchronization of Multiple Multi-Delayed Neural Networks via Impulsive Control[J]. Discrete Dynamics in Nature and Society, 2020(8):1-11.

[7] Lin W F , Tsai D Y , Tang L , et al. ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators[C]. 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, 2019: 214-218.

[8] Park J , Yoo S , Yoon S , et al. Interworking technology of neural network and data among deep learning frameworks[J]. Etri Journal, 2019, 41(6): 760-770.

[9] B. K. Reddy, S. Bano, G. G. Reddy, R. Kommineni and P. Y. Reddy, "Convolutional Network based Animal Recognition using YOLO and Darknet," 2021 6th International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 2021, pp.

[10] Liu Y, Chen C, Zhang R, et al. Enhancing the interoperability between deep learning frameworks by model conversion[C]. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020: 1320-1330.

[11] 王甜. PyTorch 至 ONNX 的神经网络格式转换的研究 [D]. 西安电子科技大学,2022.

[12] 贺凯. TensorFlow 与 ONNX 模型转换研究与实现 [D]. 西安电子科技大学,2022.

[13] 李卫华. 数字图像预处理与融合方法研究 [D]. 西北工业大学,2006.

[14] 董广军, 范永弘, 罗睿. 基于粗糙集的图像智能增强预处理 [J]. 计算机工程,2003(13):57-58+61.