CUDA/CPP 编程下的 ResNet 复现

Reproducing ResNet with CUDA and CPP

ID：	zzc18
Email1/QQ	424729227@qq.com
Email2	zzzc18@outlook.com

2022 年 6 月 25 日

目 录

第 1 章 设计选题及分析

本次课程设计我们选择“1 基于提供的卷积神经网络示例代码，进行代码的优化，例如对 GPU 并行中的一些配置参数的优化。或基于示例代码，开发 U-Net 等其他当前常用网络”，具体选择为利用 CUDA/CuBLAS/CuDNN 进行 ResNet 网络[1]的复现。

ResNet 网络的残差块结构如图 1 所示，整个网络由多个这样的残差块叠加而成，相比于无普通结构，残差块有一个输入到输出的直接加和。残差连接的引入有效缓解了梯度消失、梯度爆炸、深层网络退化等问题，使得深层网络的设计成为可能。ResNet 问世后很快受到广大科研工作者的认可，相关的跟进研究数不胜数，各类下游任务如图像分割、目标检测都会使用 ResNet 及其在 ImageNet[4]上的预训练参数作为主干网络（Backbone），并取得了很好的结果。Vision Transformer 的问世使得计算机视觉方向的 Backbone 有了更好的选择，但是根据先前 Facebook Image Similarity Challenge 2021 的比赛结果，ResNet 在数据充足的情况下可以表现出比 Swin-Transformer 更好的性能[2][3]。故对 ResNet 进行复现很有实际意义，今后如果使用 CPP 开发深度学习算法则可以在此项目的基础上进行研究。在这里我们不去分析其网络结构优秀的原因，只关注实现问题。

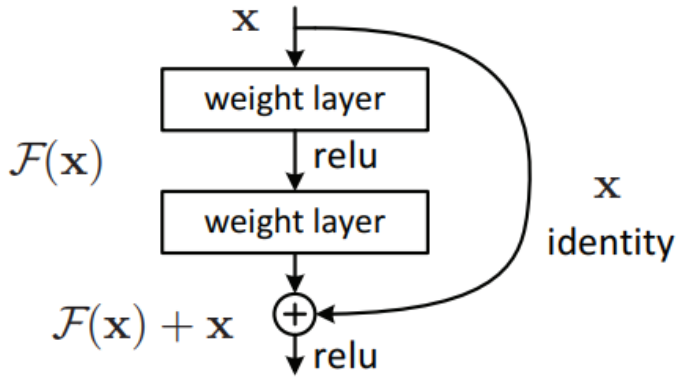

图 1 残差块结构[1]，weight layer 一般是卷积层加归一化层

从头训练参数较多的深度网络需要较大的数据集才能防止过拟合，所以我们采用 ImageNet [4]数据集 2012 年的版本，为了实现在该数据集上训练 ResNet，相比于先前的样例代码（cuda_cnn_ptr.tar.bz2 中），我们主要需要实现以下几点：

1. ImageNet 数据集的解析、预处理和数据增强
2. 框架内存和结构优化，主要是前向传播和反向传播中间结果的内存共用
3. 残差结构的引入，并引发了计算图的构建问题
4. BatchNorm 层的引入
5. ResNet 网络结构
6. 权重初始化
7. 修正线性层权重表述不当的问题
8. SGD 优化器的动量参数及权重衰减

在第二章中我们会对这些实现进行具体介绍。

第 2 章 实现过程及要点

本文中如提到代码引用，均以提交的代码文件夹作为根目录以相对路径的方式引用，由于报告撰写时涉及部分代码的注释调整，具体行数可能发生变化，故一般将引用定位到某一函数或类，文中用大括号标注代码引用，如{src/Dataset/Dataset.h:class Dataset}。同时代码也**在 GitLab 上有线上版，可能更便于参阅[7]**。

2.1 ImageNet 数据集的解析、预处理和数据增强

2.1.1 ImageNet 数据集的解析和读取

我们采用 ILSVRC2012（ImageNet 对应比赛名）数据集进行训练和验证，该版本训练集有 1281167 张图片，测试集有 50000 张图片，共计 1000 类。该数据集正确性有保障，完全可以支撑大型网络的训练，而后续年份的版本增补了较多数据，对算力要求有进一步提升，不利于我们的验证，故不作考虑。

训练数据集以文件夹形式进行分类，文件夹中有若干该类别的图片，文件夹名是其类别 ID，可以通过一个 ID 对应表对其进行识别，如图 2，3 所示。

```
(cu115) [zzzc18@zzzc18-MSI ImageNet]$ cd train
(cu115) [zzzc18@zzzc18-MSI train]$ ls
n01440764  n01824575  n02093754  n02115641  n02457408  n02877765
n01443537  n01828970  n02093859  n02115913  n02480495  n02879718
n01484850  n01829413  n02093991  n02116738  n02480855  n02883205
n01491361  n01833805  n02094114  n02117135  n02481823  n02892201
n01494475  n01843065  n02094258  n02119022  n02483362  n02892767
n01496331  n01843383  n02094433  n02119789  n02483708  n02894605
n01498041  n01847000  n02095314  n02120079  n02484975  n02895154
n01514668  n01855032  n02095570  n02120505  n02486261  n02906734
n01514859  n01855672  n02095889  n02123045  n02486410  n02909870
```

图 2 训练集文件夹结构示例

```
1000
0 n01440764 鱼, tench, Tinca tinca
1 n01443537 鱼, goldfish, Carassius auratus
2 n01484850 鱼, great white shark, white shark, man-eater,
3 n01491361 鱼, tiger shark, Galeocerdo cuvieri
4 n01494475 鱼, hammerhead, hammerhead shark
5 n01496331 鱼, electric ray, crampfish, numbfish, torpedo
6 n01498041 鱼, stingray
7 n01514668 鸡, cock
8 n01514859 鸡, hen
9 n01518878 鸵鸟, ostrich, Struthio camelus
10 n01530575 鸟, brambling, Fringilla montifringilla
11 n01531178 鸟, goldfinch, Carduelis carduelis
12 n01532829 鸟, house finch, linnet, Carpodacus mexicanus
13 n01534433 鸟, junco, snowbird
```

图 3 ID 与类别对应表示例

原始验证集直接将图像以文件形式放在文件夹中，为了与训练数据统一形式便于

解析，我们可以利用一个脚本将其进行转换[6]。

在了解其数据布局后，我们就可以较为容易地实现数据集解析。我们首先实现了 Dataset 基类{ src/Dataset/Dataset.h:class Dataset}，该类提供了一个功能函数 Glob，可以获取某文件夹下的所有内容，定义了三个纯虚函数：**Length** 用于获取数据集长度，**GetItem** 用于通过一个 int index 来获取具体数据和标签，**GetLabel** 用于通过一个 int index 来获取标签。通过这一基类我们可以统一不同数据集的形式，这样如果后续进行数据集变更则只用重写数据集解析代码，且该操作只读不写，各个图片之间没有联系，故读取时可以采用 OpenMP 较为方便地实现并行{ src/network.cu:Network::Train; src/network.cu:Network::Predict, #pragma omp}。

对于具体解析，我们用 ImageNetDataset 类{ src/ImageNetParser/ImageNetParser.h: class ImageNetDataset}继承 Dataset 类，并实现三个虚函数，对于遍历文件夹和图片名的方法在此不做赘述，图片的读取采用 OpenCV 进行操作。

2.1.2 数据预处理

由 OpenCV 读入的 JPEG 格式图片为 BGR 三通道，所以需要先通过 cv::cvtColor(img, img, cv::COLOR_BGR2RGB);进行通道重排，然后将其调整大小到 224x224，这是 ResNet 的标准输入大小，以这种大小输入的图像恰好可以满足最后原先设计的 7x7AveragePool。

经上述处理后，原始图像为 HWC（224x224x7）格式，而卷积网络一般以 CHW 格式接受图像，所以需要进行一次重新排列。此外 JPEG 格式读取后的每个像素通道的数据类型为 u_int8 格式，取值范围是[0,255]间的整数，一方面这个数值离散程度较大，另一方面其绝对值较大，直接传入网络容易导致数值溢出梯度爆炸等问题，故需进行归一化操作。对于此类大型图像数据集，我们按下式进行归一化操作，一般采用表 1 中的数值进行带入。这部分代码参考{src/ImageNetParser/ImageNetParser.h: HWC2CHW}。

$$x_i' = \frac{x_i - E(x)}{Var(x)}$$

表 1 图像归一化所用参数

通道	R	G	B
均值 E(x)	0.485	0.456	0.406
方差 Var(x)	0.229	0.224	0.225

2.1.3 数据增强

数据增强通过对图像或其他数据施加某些不改变其类别信息的变换，如长宽比，对比度等，可以让有限的数据产生更多的数据，增加训练样本的数量以及多样性（噪声数据），提升模型鲁棒性，防止模型出现过拟合。在我们的实验中，数据增强后训练精度收敛速度变慢，上界也低于无增强的情况，但是可以使验证集精度与训练集相当，说明有良好的防止过拟合的效果。本小节相关代码可以参考{src/ImageNetParser/Transforms.h}，部分代码的实现参考了[11]中的实现。

在我们的实验中，主要用到两种数据增强方法，随机水平翻转和随机裁剪，随机

水平翻转这里不再解释，我们介绍一下随机剪裁的方式。随机剪裁的代码在类{src/ImageNetParser/Transforms.h:class RandomResizedCrop}中，其分为两个步骤，首先随机获取一块指定区域，其次将其 Resize 到指定大小（实验中即为 224x224）。随机获取区域有两个参数 scale 和 ratio：scale 指定一个范围，如 0.08 到 1，表示随机选取的区域面积大小与原始图像大小的比值；ratio 也是一个范围，如 0.75 到 1.33，表示选取区域的长宽比。在指定了长宽后可以选取一个合适的起始点坐标作为锚点，用公式可以表示为：

$$
\begin{aligned}
S &= hw \\
S' &= scale \times S = h'w' \\
h' &= \sqrt{S' \times ratio} \\
w' &= \sqrt{S'/ratio} \\
x &\in [0, h - h'] \\
y &\in [0, w - w'] \\
BoundingBox &= (x, y, h', w')
\end{aligned}
$$

不难发现，这样选取的区域满足上述条件，但不一定可以从原始图像中选取出来，比如计算得到的h'要大于原始图像高度h，因此我们随机做十次上述操作，如果某一次满足选区在原始图像大小范围内则进行裁剪；如果 10 次后仍没有满足条件的选择范围，我们选取原始图像作为裁剪选区。

为了防止每次随机分布都是以同一个 seed 产生的第一个随机数来进行生成，我们用一个单一实例类 RandomGenerator 来产生一个全局的随机数生成器，其代码可以参考{src/ImageNetParser/Transforms.h:class RandomGenerator}。

2.2 框架内存优化

除特殊说明，本节内存与显存表示同一含义。图例中对于 Weight/Bias & Grad 的表述可以理解为指针，即部分节点如 Pool，Activation，Split，Residual，Conv 的 Bias 等不具有权重，但该指针仍存在，图中标注不代表内存中也分配有空间，仅供读者参考。

2.2.1 数据和梯度显存共用

在原始框架下，如果直接实现 ResNet50 模型，在 BatchSize 为 32 的情况下，其将消耗近 14GB 的显存，而相同情况下 PyTorch 只需要近 8GB 显存，经过对显存占用的分析，我们发现权重及其梯度的空间占用较小，总计占用不到 200MB 显存，而记录中间计算结果的部分占用了 4600MB 的显存，而其梯度所占用空间和其一样大，而这部分内存实际上可以通过一个缓冲区进行复用。

在原先给定的框架中，每个节点（实际为模型中的一层，但这样便于描述，下同）在内存中由如图 4 的部分组成。对于所有带权重的节点（主要是卷积层、归一化层和线性层等），其前向传播可以表示为（式中变量为矩阵）：

$$Y = XW + B$$

而反向传播时，若代价函数为 J ，且假定 $\frac{\partial J}{\partial Y}$ 已知，则可归纳地表示为：

$$\frac{\partial J}{\partial X} = \frac{\partial J}{\partial Y} W^T$$
$$\frac{\partial J}{\partial W} = X^T \frac{\partial J}{\partial Y}$$
$$\frac{\partial J}{\partial B} = \sum_{b}^{N} \frac{\partial J}{\partial Y}$$

其中$\frac{\partial J}{\partial X}$将是前一层的$\frac{\partial J}{\partial Y}$。

若设学习率为 η，则权重更新可表示为：

$$W = W - \eta \frac{\partial J}{\partial W}$$
$$B = B - \eta \frac{\partial J}{\partial B}$$

在上述公式的基础上，可以发现反向传播时当前节点的输入数据 X 只会影响当前节点的权重梯度矩阵的计算，不影响前一个节点的梯度求解，且在参数更新时不起作用。也就是说**如果一个节点的梯度已完成计算，那么其输入数据可以清空而不影响计算结果**，而数据和数据梯度的大小恰好相同，即图 4 中的 Input Data 和 Input Data Grad 部分，这就带来了复用的空间。由于 CuDNN/CuBLAS 对大部分节点不支持 InPlace 操作，我们需要额外开辟一个缓冲区来储存 $\frac{\partial J}{\partial X}$ 的结果，当前节点计算完成后将其拷贝到 Input Data 区域作为输入的梯度，这个缓冲区全局只需要一个，其大小为中间计算结果所占空间的最大值。这样，我们的节点就可以优化为图 5 的形式。具体对缓冲区的使用可以参考如 {src/convolutional_layer.cu:Conv2D::Backward} 中 d_temp_grad_features_ 用于临时保存计算的梯度值，以及 {src/layer.cu:Layer::BackwardCopy}将 d_temp_grad_features_再拷贝到 Input。

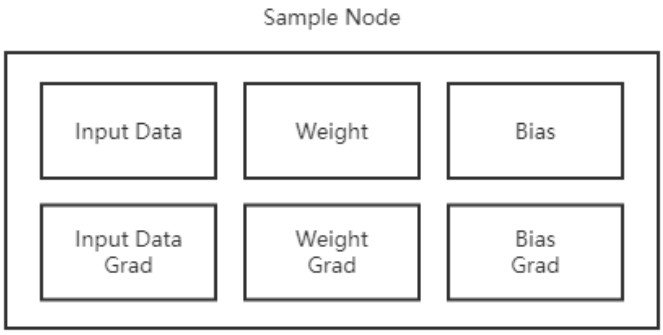

图 4 原始节点结构示例

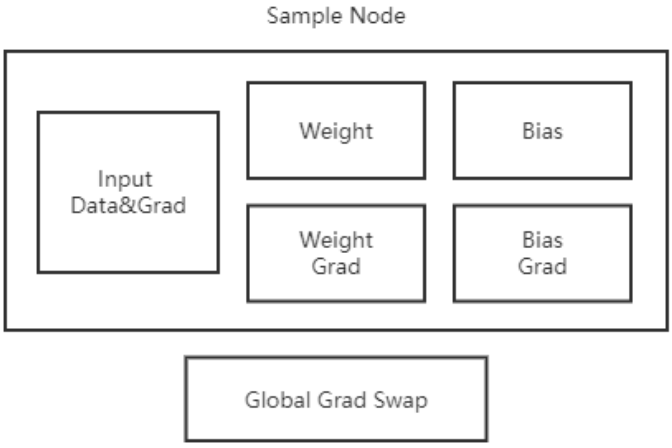

图 5 改进节点结构示例

优化后，ResNet50 在 BatchSize 为 32 时占用约 9GB 显存，BatchSize 为 64 时占用约 18GB 显存，有明显改进，而与 PyTorch 仍存在一定差距，这一内容会在第 5 章进行分析。

2.2.2 内存分配结构优化

本小节还是针对数据和数据梯度的分配，首先对节点内存分配方式进行介绍。网络初始化时，每个节点会计算自己所需的输入和输出数据空间的大小，然后网络对这些空间大小求和并利用指针给每个节点分配内存，一个很重要的性质是**当前节点的输出内存和下一个节点的输入内存是一块空间**，即节点之间构成链表结构。原先网络以每一层的输入作为基准标志，即前一层的输出空间会连接到当前层的输入，而我们将其改变为以输出层作为当前层的基准，令下一层的输入连接到本层的输出。这两种分配方式在内存上没有区别，但在每一个 Layer 类构建时，相当于默认分配输出空间而非输入空间。这种方式更便于理解，且在后续的残差结构中起到了作用。节点分配方式如图 6 所示，代码参考{src/network.cu:Network::AllocateMemoryForFeatures}。

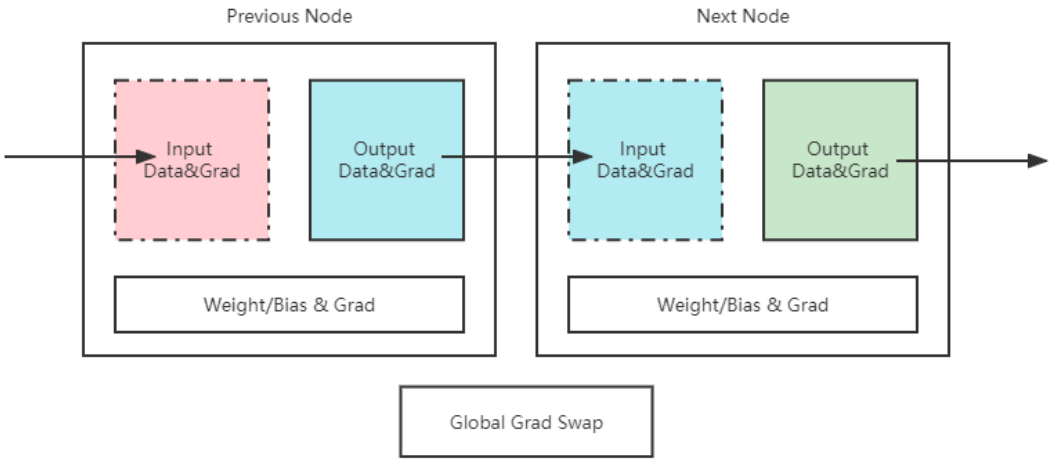

图 6 节点分配方式

相同颜色为内存上同空间的部分，实线框为基准空间

2.3 残差结构与计算图

2.3.1 计算图

由于如图 1 的残差结构的引入，原先的单链式网络结构不再适用，网络有了分支节点与合并节点的概念。为了让前向传播和反向传播的梯度计算不错乱，需要引入一个有向无环图（DAG）作为计算图，利用拓扑排序获得正确的计算顺序然后再依次计算。

LayerGraph 类{src/layer.h:class LayerGraph}以 Layer 类的指针作为节点编号，用临界表保存了计算图的正向和反向连接图，函数{src/layer.cu: LayerGraph::AddEdge}实现了加边，函数{src/layer.cu:LayerGraph::TopoSort}实现了对计算图的拓扑排序，拓扑排序的结果保存在该类下的 std::vector<Layer *> layers_; 中。有专门的类来管理建图，我们就不需要按顺序添加节点，只需要用 LayerGraph::AddEdge 将其连接好，指定好顺序即可，这也大大提高了网络的可编程性。其调用方式可以参考{src/resnet.cu:ResNet::AddLayers}。

下面以一个例子说明拓扑排序的作用，以图 7 为例，如果按构造节点的顺序进行调用，则可能会以 Input-Split-Conv1-BN1-ReLU-Conv2-BN2-ReLU-Residual-Conv Downsample-BN DownSample-Output 的顺序调用，这样 Residual 在处理加和时其中一个输入还没有准备好，结果会是有问题的，而经过拓扑排序后我们可以给出一个形如 Input-Split-Conv1-BN1-ReLU-Conv2-BN2-Conv Downsample-BN DownSample-ReLU-Residual-Output 的调用序列（不唯一），这样就能保证每个节点执行时其输入都是准备好的。

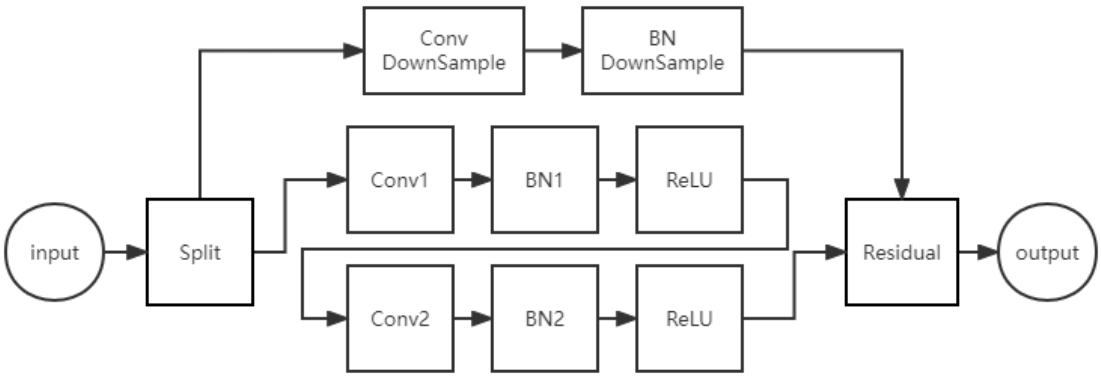

图 7 ResNet 中带 DownSample 节点的 BasicBlock

2.3.2 分裂节点和合并节点

为了实现如图 1 的残差结构，我们在网络中引入了两种特殊节点，分裂节点和合并节点。对于 ResNet 中的 BasicBlock 结构，如图 8 所示，如果不构造新的节点直接利用先前构造，残差结构的实现将较为复杂，所以我们引入这两种额外节点来处理这个问题，如图 9 所示。2.3.3 和 2.3.4 节将具体描述实现细节。此外，不难发现有了这两类节点我们可以依次构造任意多分支的网络，不过多分支结构应对应多输入/输出才更高效，这点在第 5 章讨论。

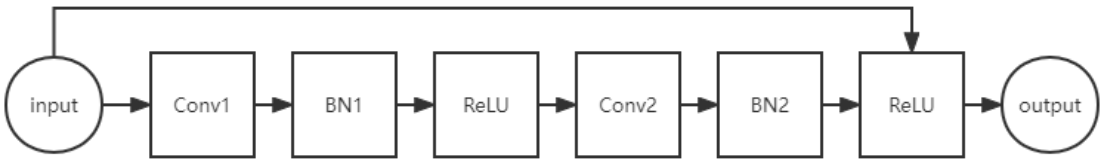

图 8 原始 BasicBlock 结构

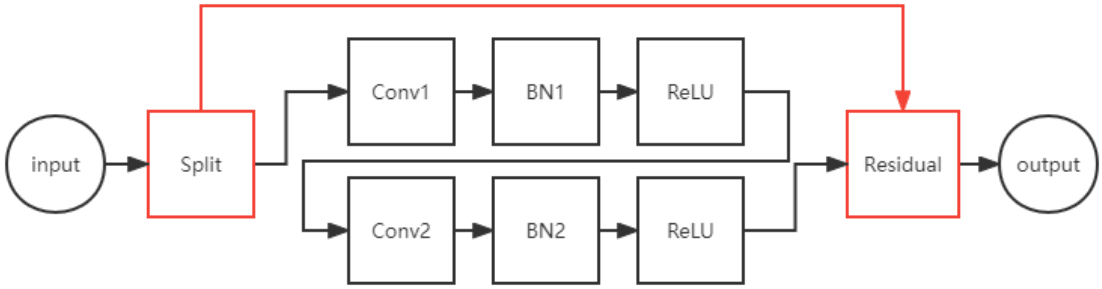

图 9 增加分裂节点 Split 与合并节点 Residual 的 BasicBlock 结构

2.3.3 分裂节点

分裂节点，即类 Split{src/residual_layer.h:class Split}有一个输入两个输出，其结构如图 10 所示。其前向传播只负责拷贝前一层的输出到自己的输入，即将图 10 中的红色部分拷贝到蓝色部分；反向传播时较为特殊，分裂节点需要**对后续节点的梯度进行求和**，所有分裂节点的 Output 要在反向传播时被清零以便累加梯度，而处在在分裂节点后的节点，如图 10 中 Next Node 1 和 Next Node 2 位置的点，在求导时需要把分裂节点的 Input 作为自己的 Input。以图 10 的 Next Node 1 为例，对于公式：

$$\frac{\partial J}{\partial W} = X^T \frac{\partial J}{\partial Y}$$

其中的$\frac{\partial J}{\partial W}$是白色部分的权重梯度，$\frac{\partial J}{\partial Y}$ 是绿色部分，X^T**是红色部分而不是蓝色部分**（对于不在分裂节点后的节点则是蓝色部分）。对于分裂节点，前向传播的实现在{src/residual_layer.cu:Split::Forward}；而反向传播的实现则不仅体现在{src/residual_layer.cu:Split::Backward}，还体现在所有在求导时对输入数据有利用的节点的 Backward 的前几行，可以参考{src/convolutional_layer.cu:Conv2D::Backward}中的前几行（摘录在下方），此外还体现在{src/layer.cu:Layer::BackwardCopy}中对分裂节点梯度的累加。

```
float *xPtr = input_.CudaPtr();
if (previousSplitLayer_ != nullptr) {
    xPtr = previousSplitLayer_->GetInput().CudaPtr();
};
```

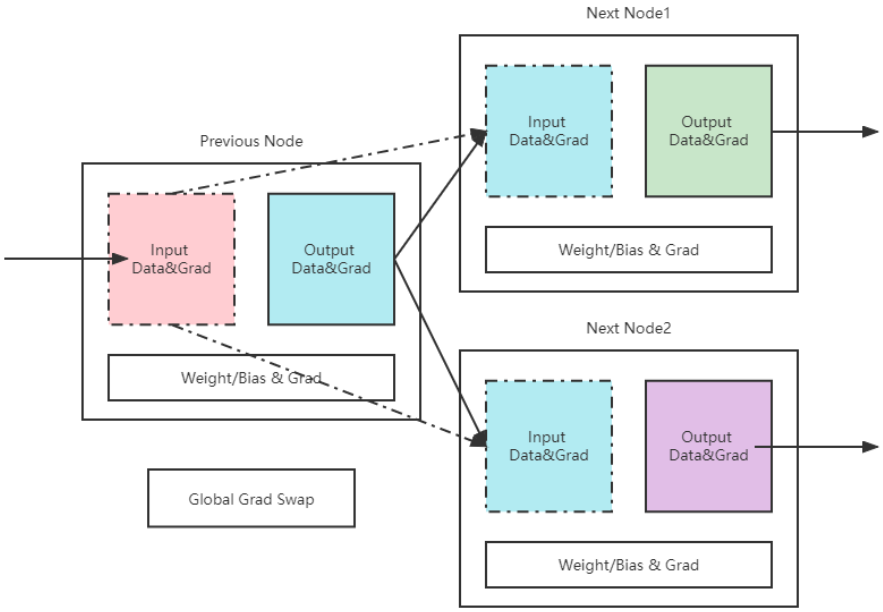

图 10 分裂节点示例

2.3.4 合并节点

合并节点，即类 Residual{src/residual_layer.h:class Residual}有两个个输入一个输出，其结构如图 11 所示。其前向传播中不使用 Input 指针，直接以前一层的两个节点的输出作为输入，将其求和后作为自己的输出；反向传播也较为简单，可以直接将自己的输出梯度拷贝至两个输入节点的输出梯度。

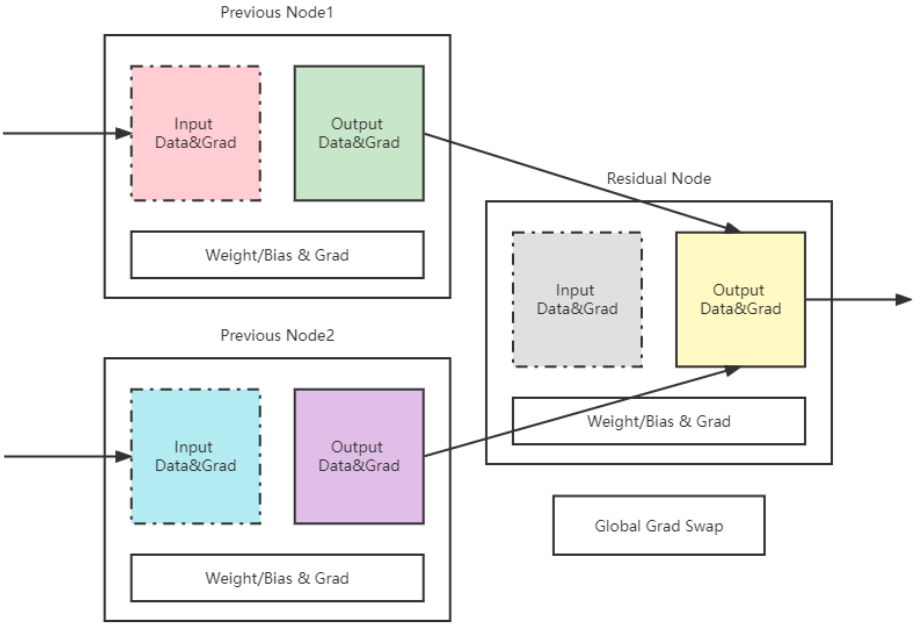

图 11 合并节点示例

2.4 BatchNorm 层引入

BatchNorm 层通过对每一个 Batch 的特征向量进行归一化，使其统计量仍为均值为 0 方差为 1 的状态，可以使网络更快速的收敛，减少由于数值累积造成的溢出，并且通过滑动平均来保留一些先前 Batch 中的统计信息，相当于增大 BatchSize，增加模型泛化性[8]。用数学公式可以表示为：

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}\,[x] + \epsilon}}\gamma + \beta$$

其中 γ 和 β 是可学习的参数，其大小为输入输出通道数的大小，ϵ 用于防止出现方差为 0 或过小的问题，一般取 1E-5。除此之外，BatchNorm 还用指数滑动平均维护历史的统计量，即：

$$\hat{x}_{t+1} = (1 - \mu)\hat{x}_t + \mu x$$

其中 x 为方差或均值，μ为滑动平均参数，通常取 0.1，BatchNorm 在训练时直接采用当前 Batch 的均值和方差进行计算，在测试时则使用历史统计量[9]

具体在实现方面，我们直接调用 CuDNN 库实现的 BatchNorm **错误!未找到引用源。**，使用 CUDNN_BATCHNORM_SPATIAL 模式，与此同时，由于 BatchNorm 层相当于带缩放和偏置的 Bias 权重，故其前一层的 Conv 不再使用 Bias，代码参考{src/batchnorm_layer.cu}。

2.5 ResNet 网络结构

ResNet 网络结构层次较为分明，大致可以分为以下几部分，我们以输入大小为 Batchsize=64，图像大小为三通道 224x224 的标准输入为例介绍该网络结构（[N,C,H,W]=[64,3,224,224]）。代码参考{src/resnet.cu:ResNet::AddLayers}。网络结构的代码我们参照 PyTorch 子模块 torchvision 中的描述进行编写[5]。

2.5.1 基本结构，BasicBlock 与 Bottleneck

本小节代码可以参考 {src/resnet.cu:ResNet::AddBottleneckBlock} 以及 {src/resnet.cu:ResNet::AddBasicBlock}。

首先对于网络中的卷积节点，一般都以 **Conv+BN+ReLU** 的形式出现，可以理解为这三者组合在一起是一个最小单位，但是如果残差连接需要下采样时可能两个卷积节点会共用一个 ReLU 节点。结构中出现的所有卷积都使用 Same Padding，即不会因 Padding 问题影响输入输出大小，卷积核大小为 7 时 Padding 为 3，大小为 3 时 Padding 为 1，大小为 1 时 Padding 为 0。网络中更改输出长宽的方式只有 Stride=2 的卷积以及池化操作。

ResNet 由若干结构相同的 BasicBlock 与 Bottleneck 堆叠而成，Basicblock 的基本形式如图 9 所示，带下采样的形式如图 7 所示；Bottleneck 的基本结构如图 12 所示，带下采样的形式如图 13 所示。每个 Block 的参数有输出维数大小 planes，卷积 stride 值，对于 BasicBlock：

Conv1=[Kernel=3, Padding=1, OutDim=planes, Stride=stride],

Conv2=[Kernel=3, Padding=1, OutDim=planes, Stride=1]；

对于 Bottleneck：

Conv1=[Kernel=1, Padding=0, OutDim=planes, Stride=1],

Conv2=[Kernel=3, Padding=1, OutDim=planes, Stride= stride],

Conv3=[Kernel=1, Padding=0, OutDim=planes*4, Stride=1]。

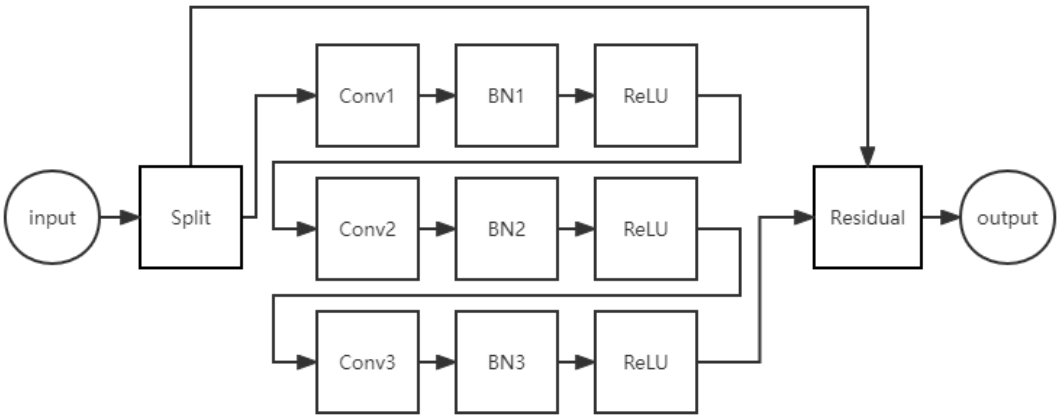

图 12 ResNet 中的 Bottleneck

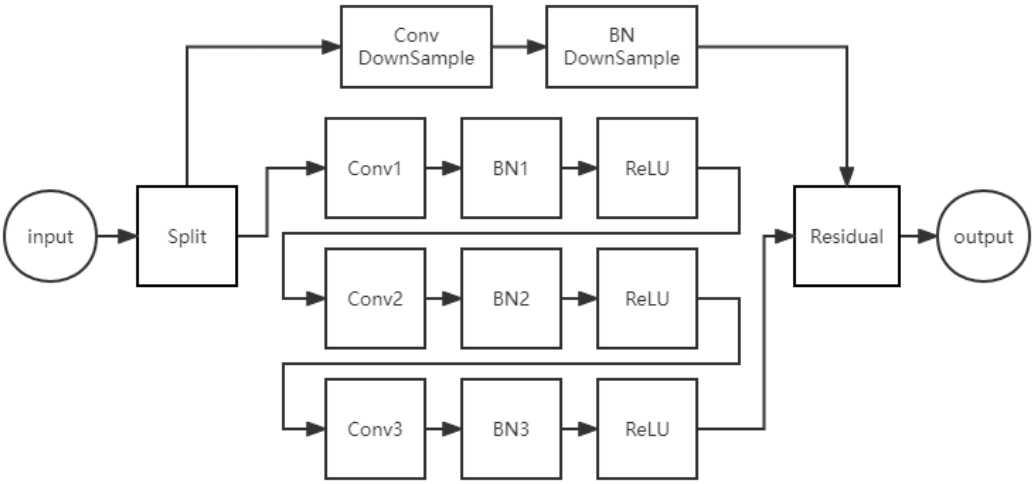

图 13 ResNet 中带 DownSample 节点的 Bottleneck

2.5.2 输入浅层网络（Stem 层）

输入层由 Conv，BN，ReLU 和 MaxPool 组成，其中：

Conv=[Kernel=7, Padding=3, OutDim=64, Stride=2],

MaxPool=[Kernel=3, Stride=2],

输入图像/特征通过各层后的形状变化为[64,3,224,224]--[64,64,112,112]--[64,64,112,112]--[64,64,56,56]。

2.5.3 四层结构

在经过输入层后，特征会依次经过 4 层结构类似参数不同的 Layer，每个 Layer

包含若干 Block，即 2.5.1 中的 BasicBlock 或 Bottleneck，其具体参数和输出维度如表 2 所示。对表中数据的解读，以 50 层结构为例，在经过第一层 Layer 后，其输出维数为[64,256,56,56]，在经过第二层 Layer 后，其输出维数为[64,512,28,28]，以此类推。这部分代码可以参考{src/resnet.cu:ResNet::MakeLayer}。

2.5.4 输出池化与全连接层

对于标准大小的输入，在经过四层结构后其大小应为[64,*,7,7]，其中*在使用 BasicBlock 时为 512，在使用 Bottleneck 时为 2048。随后我们将其经过一个 Average Pool=[Kernel=7, Stride=1]，将其输出大小转变为[64,*,1,1]，并在其后跟全连接层用于输出分类，全连接层参数矩阵大小为[*,ClassNum]，ClassNum 为分类类别数，对于 ImageNet 是 1000。如果原始输入大小的长宽不是 224x224，我们也一并将其最终池化后的长宽变为 1x1，这在测试时有助于处理不同分辨率的图像，我们可以在测试时选择分辨率略高的图像来提升精度。

表 2 四层结构对应块及参数表[1]

18 层和 34 层使用 BasicBlock，34 层及以上使用 Bottleneck

Conv2_x 对应第一个 Layer，往后递增，以此类推

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

2.6 权重初始化

在调试的过程中，我们发现特征的统计量（均值和方差）随着层数的增加存在不稳定的现象，且这一现象在权重改变前就已经出现，说明该现象很可能与初始化有关。好的初始化可以有效提高模型收敛的速度，不好的初始化甚至可能使其不收敛。

对于卷积层和线性层，我们将网络的初始化方式改为 Kaiming Init[15]，这种初始化方式对 ReLU 激活方式的卷积层更为友好，在使用后我们也发现各层统计量的数量级基本保持一致，没有过大或过小值出现。这种初始化方式可以具体表述为：

$$W_{CONV} \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$
$$n^l = OutChannel \times KernelSize^2$$
$$W_{FC} \sim U\left[-\frac{1}{\sqrt{InDim}}, \frac{1}{\sqrt{InDim}}\right]$$
$$B_{FC} \sim U\left[-\frac{1}{\sqrt{InDim}}, \frac{1}{\sqrt{InDim}}\right]$$

上述内容代码位于{src/layer.cu:Layer::InitiateWeightsAndBiases}。

对于 BatchNorm 层$y = \frac{x-\mathrm{E}[x]}{\sqrt{\mathrm{Var}\,[x]+\epsilon}}\gamma + \beta$我们分两种情况处理，首先一般情况下我们将$\gamma$初始化为全 1，$\beta$层初始化为全 0；其次如果 BatchNorm 层处于一个残差连接前，即如图 7 中的 BN2 或图 13 中的 BN3，则将其γ初始化为全 0，根据[16]，这么做有利于更贴近残差网络的设计理念，即让特征先通过跳跃连接直接向后传播，再让中间的部分学习需要进一步细化的内容。对于所有的 BatchNorm 层，我们将其历史的 $\tilde{E}[x]$ 初始化为全 0，$\widetilde{Var}[x]$ 全部初始化为 1，上述代码位于{src/batchnorm_layer.cu:Batchnorm2D::InitiateWeightsAndBiases}。

2.7 修正线性层权重表述不当

我们发现原始的线性层的计算存在问题，原始线性层权重矩阵与特征矩阵相乘的代码如图 14 所示，按照图 14 中的描述，线性层以$Y = XW + B$的形式进行计算，而根据先前分配的内存以及卷积层的 CuDNN 输出描述符，我们可以知道（以 ResNet18 的输出维度 512 为例，下同）：

$$W \in \mathbb{R}^{[512\times1000]}$$
$$X \in \mathbb{R}^{[n\times512]}$$
$$n = BatchSize$$

考虑到 **CuBLAS 对保存矩阵的连续内存解析为列优先而不是行优先**，原先的输入矩阵和输出结果在 CuBLAS 看来应该是：

$$W^T \in \mathbb{R}^{[1000\times512]}$$
$$X^T \in \mathbb{R}^{[512\times n]}$$
$$Y^T = (XW)^T = W^T X^T \in \mathbb{R}^{[1000\times n]}$$

此时的Y^T在内存中的排布恰满足$Y^T \in \mathbb{R}^{[n\times1000]}$，那么描述符 CUBLAS_OP_T 就多余了，虽然如不考虑形状问题只从内存角度出发该计算结果没有问题，但保存时如按原始形状保存则权重矩阵与其他框架将无法匹配。

```
checkCublasErrors(cublasSgemm(cuda_->cublas(), CUBLAS_OP_T, CUBLAS_OP_N,
                              output_shape_, batch_size, input_shape_,
                              &cuda_->one, weights_.CudaPtr(), input_shape_,
                              input_.CudaPtr(), input_shape_, &cuda_->zero,
                              output_.CudaPtr(), output_shape_));
```

图 14 原始线性层前向传播代码

而对于反向传播，根据上述公式及 2.2.1 节中的公式推导，我们知道：

$$\frac{\partial J}{\partial W} = X^T \frac{\partial J}{\partial Y}$$

$$X \in \mathbb{R}^{[n\times 512]}$$

$$\frac{\partial J}{\partial Y} \in \mathbb{R}^{[n\times 1000]}$$

$$\frac{\partial J}{\partial W} \in \mathbb{R}^{[512\times 1000]}$$

在 CuBLAS 看来，我们的矩阵和计算实际上应是：

$$(\frac{\partial J}{\partial W})^T = (X^T \frac{\partial J}{\partial Y})^T = (\frac{\partial J}{\partial Y})^T X$$

$$X^T \in \mathbb{R}^{[512\times n]}$$

$$(\frac{\partial J}{\partial Y})^T \in \mathbb{R}^{[1000\times n]}$$

$$(\frac{\partial J}{\partial W})^T \in \mathbb{R}^{[1000\times 512]}$$

而原始代码如图 15 所示，在考虑到权重矩阵描述与实际不符的情况下仍有正确性，但形式上与公式推导有一定出入。

```
cublasSgemm(cuda_->cublas(), CUBLAS_OP_N, CUBLAS_OP_T, input_shape_,
            output_shape_, batch_size, &cuda_->one, input_.CudaPtr(),
            input_shape_, output_.CudaPtr(), output_shape_, &cuda_->zero,
            grad_weights_.CudaPtr(), input_shape_);
```

图 15 原始线性层反向传播代码

我们将前向传播和反向传播修正为满足权重矩阵形状的形式，前向传播如图 16 所示，反向传播如图 17 所示，按这种方式实现完全符合上述公式的推导。

```
// output = weightsT * input (without biases)
checkCublasErrors(cublasSgemm(
    cuda_->cublas(), CUBLAS_OP_N, CUBLAS_OP_N,  // OP(A),OP(B)
    output_shape_, batch_size, input_shape_,    // m,n,k
    &cuda_->one,                                // alpha
    weights_.CudaPtr(), output_shape_,          // A, lda
    input_.CudaPtr(), input_shape_,             // B, ldb
    &cuda_->zero,                               // beta
    output_.CudaPtr(), output_shape_            // C, ldc
    ));
```

图 16 修改后线性层前向传播代码

```
// dw = xT * (dy)
checkCublasErrors(cublasSgemm(
    cuda_->cublas(), CUBLAS_OP_N, CUBLAS_OP_T, output_shape_, input_shape_,
    batch_size, &cuda_->one, output_.CudaPtr(), output_shape_, xPtr,
    input_shape_, &cuda_->zero, grad_weights_.CudaPtr(), output_shape_));
```

图 17 修改后线性层反向传播代码

2.8 SGD 优化器的动量参数及权重衰减

本节代码可以参考{src/network.cu:Network::Update}，相关实现参考了[12]。

随机梯度衰减算法（Stochastic Gradient Descent, SGD）的动量（Momentum）优化也不仅使用当前步骤的梯度来指导搜索，而是累积过去步骤的梯度以确定要去的方向，引入动量就可以加速我们的学习过程，可以防止模型陷入鞍点，也可以理解为某种意义上的增大 Batchsize，其具体实现过程就是当前的梯度值为历史梯度值的指数滑动平均。

权重衰减则是通过将权重的平方的一半加至最终的代价函数，用公式表述就是：

$$J(\theta) = J_0(\theta) + \lambda \sum \frac{1}{2}\theta_i^2$$

$$\frac{\partial J(\theta)}{\partial \theta_i} = \frac{\partial J_0(\theta)}{\partial \theta_i} + \lambda \theta_i$$

通常λ的取值很小，引入权重衰减后相当于惩罚过大的权重值，从而让各特征空间有相对均匀的表述，防止模型出现过拟合。不是所有参数都适用权重衰减，对各层的 Bias 项以及 BatchNorm 的 weight 做权重衰减是没有意义的，所以一般只对参与乘法的权重做衰减，如卷积核、线性层的 weight 等。

优化器整体算法流程如图 18 所示。

2.9 其他实现

2.9.1 标签平滑（Label Smoothing）

标签平滑就是把 one-hot 形式的目标向量中 1 的值略微减小，0 的值变为一个较小数，使其和仍为一，用公式表述为：

$$\begin{cases} y_i = 1 & i = label \\ y_i = 0 & i \neq label \end{cases} \Rightarrow \begin{cases} y_i = 1 - \epsilon & i = label \\ y_i = \epsilon/(ClassNum - 1) & i \neq label \end{cases}$$

标签平滑可以防止模型对某一分类过于“自信”，可以增强模型的泛化能力，在分类模型的数据增强使用随机剪裁时，该方法也可以缓解剪裁部分不包含类别信息从而导致错误训练的问题，此外标签平滑还可以让同类特征在向量空间上更为相近[13]。从信息论的角度看，标签平滑提供了更多信息量，有助于模型学习到更多信息。其缺点是对负类不加选择地使用了同样的标签值，有阻碍模型精度提升的可能。

2.9.2 参数保存

为了更方便地观察权重以及调试，我们通过第三方工具[14]实现了将各部分参数保存为 NumPy 格式，代码位于{src/blob.h:BlobPointer::SaveAsNumpyArray}。

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1 - \tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_{t-1} + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

图 18 动量及参数衰减伪代码[12]
我们只实现了图中的 momentum 和 weight deacy 部分

2.9.3 数据排列顺序随机化

原始样例代码中给出了 rand_perm 映射数组的声明但没有使用，对于 ImageNet 数据集或其他大数据集，如果不经过随机处理一个 Batch 很可能全是同一类的信息，这种情况下迭代一次就可以使模型坍缩，我们将随机化排列做了完整的实现，代码位于{src/network.cu:Network::Train}中含 rand_perm 的部分。

2.9.4 修正 setZero<<<grid, block>>>错误调用

样例代码的{src/network.cu}中有此错误调用，该函数可以清零一段连续空间，但样例代码中反复调用该清零函数若干次，对相同的区域进行反复清零，在网络规模和数据集较小时无法体现，在我们的调试过程中发现该过程时长甚至为一分钟以上。**如果老师有意修正该错误可以查看样例代码压缩包中此部分代码。**

2.9.5 增加了 CrossEntropy Loss 具体数值的计算

Softmax+CrossEntropy Loss 反向传播时不需要计算具体代价函数值，计算该值有助于判断模型收敛情况，代码位于{src/softmax_layer.cu:CrossEntropyLoss}有多个同名函数是为了适应不同的输入格式。

第 3 章 实验结果及分析

我们的实验结果都在同一台机器上进行，机器配置为：操作系统 Manjaro，CPU Intel i9-10900K，GPU NVIDIA GeForce RTX 3090(24GB)，内存 64GB。对于各类实验，我们统一采用如下配置：

1、BatchSize 为 64

2、初始学习率为 0.1，SGD 优化器的动量为 0.875，权重衰减为 1/32768，其中对部分参数按 2.8 节的描述不做梯度衰减。

3、使用足够多的 CPU 线程读取数据以避免 GPU 出现瓶颈现象

4、如果不加标明，学习率每个 Epoch 衰减至原先的 0.975 倍，若标明按阶梯式（StepLR）调整，则每 15 个 Epoch 学习率缩减为原先的 0.1 倍。

3.1 ResNet18 实验结果

ResNet18 可以用于网络结构正确性、BasicBlock 正确性、前向和反向传播以及参数更新正确性的验证。我们在 ResNet18 上做了四组配置不同的实验，其迭代次数可能不相同，我们在观察到验证集准确率长时间不上升或下降后便手动停止以便节省时间。

3.1.1 训练 Loss

训练时的 Loss 下降曲线如图 19，图 20 所示，可以发现使用数据增强后训练 Loss 明显，使用指数下降方式可以让训练 Loss 下降相对更快，也更平滑，但训练 Loss 与验证集准确率不是完全正相关的。是否使用标签平滑在训练 Loss 上体现不明显。

3.1.2 训练准确率与测试准确率

训练集和验证集准确率曲线如图 21，22 所示，可以发现没有数据增强的方法虽然有很高的训练集准确率，但出现了明显的过拟合现象；增加了数据增强后过拟合问题基本得到解决。对照是否使用阶梯式学习率（StepLR），我们发现使用 StepLR 将更有利于模型的收敛，且各阶段的学习率相对来说更为可控，模型总是在进入平台期后再缩减学习率。

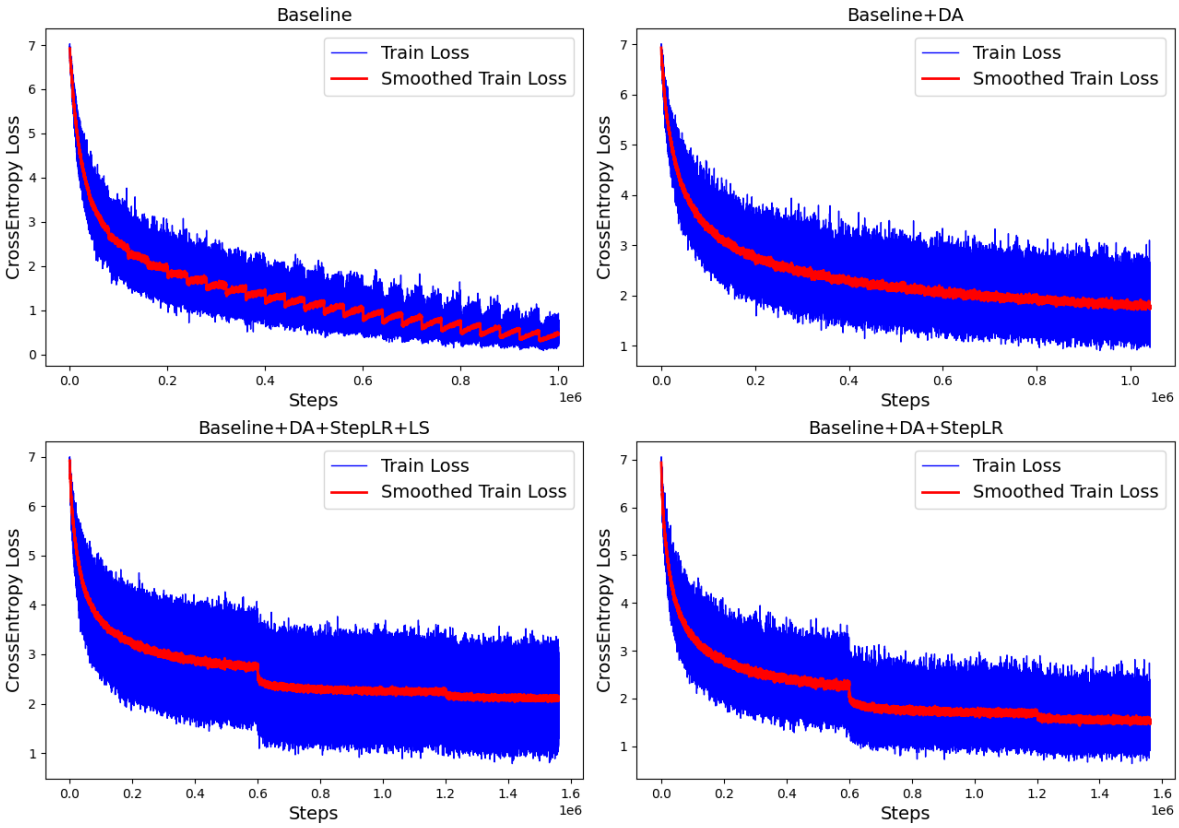

图 19 ResNet18 训练 Loss 曲线

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

蓝色曲线为原始各个 Batch 的 Loss，波动较大；

红色曲线是对一定区间做平滑处理的结果，可以反应 Loss 下降的整体趋势。

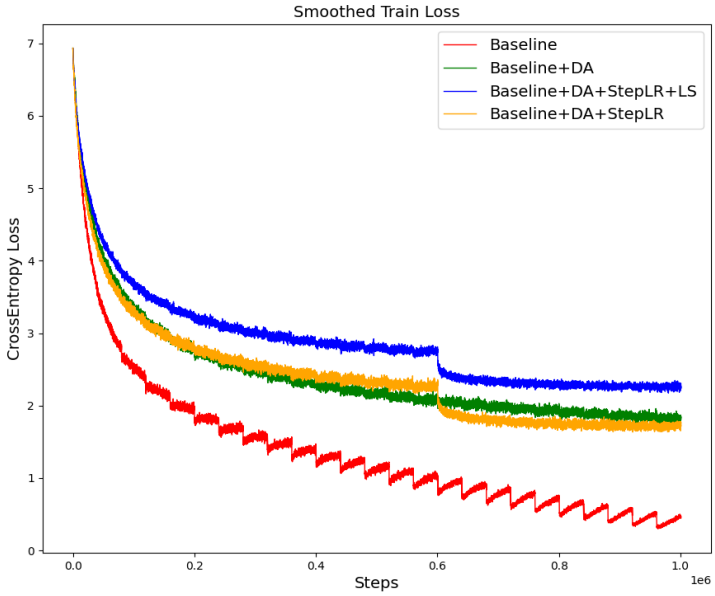

图 20 ResNet18 训练 Loss 曲线比较图

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

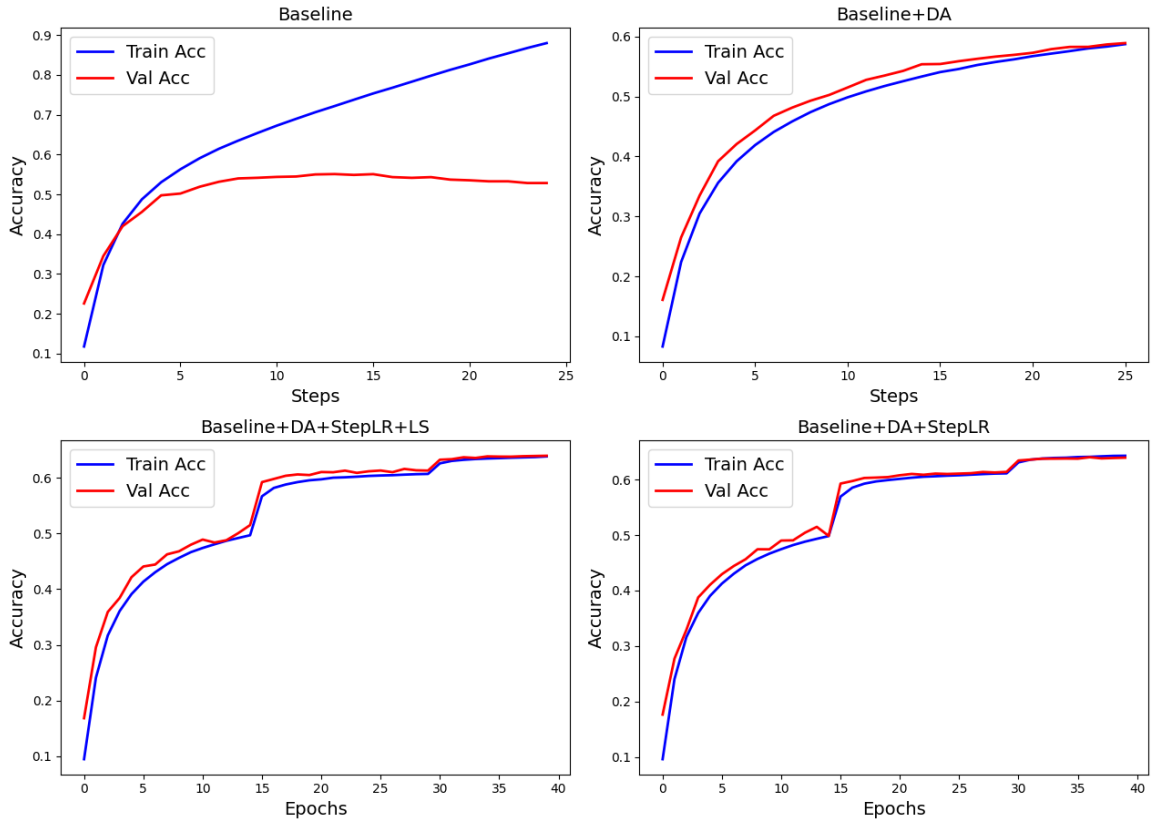

图 21 ResNet18 的训练集和测试集准确率

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

蓝色为训练集准确率，红色为验证集准确率

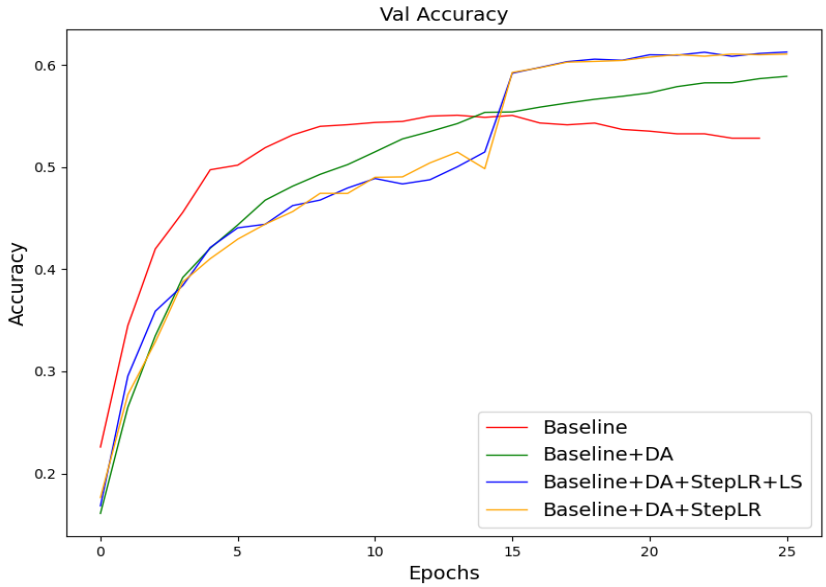

图 22 ResNet18 的测试集准确率比较（25Epoch）

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

3.1.3 结果分析

我们的实验结果如表 3 所示，其中添加了 PyTorch 提供的预训练参数测试的结果。至于表中结果可能与网络中流传的结果以及[1]中有较大出入，是因为他们普遍使用了诸如 10-Crop 等测试时增强的方法，其无增强实验结果应与 PyTorch 的结果以及我们的结果相近。

表 3 的结果说明我们的实现具有正确性，在相同的测试方式下，其正确率与 PyTorch 相差约 1%，这 1%的产生可能有多种因素，如我们的 Batchsize 较小、随机化种子不同、优化器及参数不同、迭代次数不足、训练数据集或许不同等等，但是我们认为这一结果在误差允许范围内，足够说明我们的实现的正确性。由于 C++常数较小，且我们的内存分配较为连续，我们在训练时的速度略快于 PyTorch，在测试时由于我们没有做优化，如内存管理、计算优化等，且我们统计了 Confusion Matrix，故我们的速度略慢于 PyTorch。

表 3 的结果还表明了数据增强在训练 ImageNet+ResNet 中的重要意义；阶梯式学习率对 SGD 学习有益；以及标签平滑在我们的实现中意义不大，这可能是分类过多造成的干扰项太少的缘故。

表 3 ResNet18 实验结果

方法	验证集准确率 Val Acc	单 Epoch 训练用时（秒）	单 Epoch 验证用时（秒）
Baseline	55.0%	1085	30
Baseline+DA	58.9%*	1086	30
Baseline+DA+StepLR+LS	64.51%	1083	30
Baseline+DA+StepLR	64.40%	1085	30
PyTorch	65.61%	1150	27

*: 30Epoch 的结果，该方式准确率增长缓慢，可能在超长训练后能达到更高精度，鉴于实验条件和时间不允许我们不再做验证

3.2 ResNet50 实验结果

ResNet18 可以用于网络结构正确性、Bottleneck 正确性、前向和反向传播以及参数更新正确性的验证。由于参数较多，所需训练时间较长，我们只做了两组实验，其中第一组因为出现了过拟合现象被我们提前终止。

3.2.1 训练 Loss

训练时的 Loss 下降曲线如图 23，图 24 所示，相关性质表述同 3.1.1 中内容。

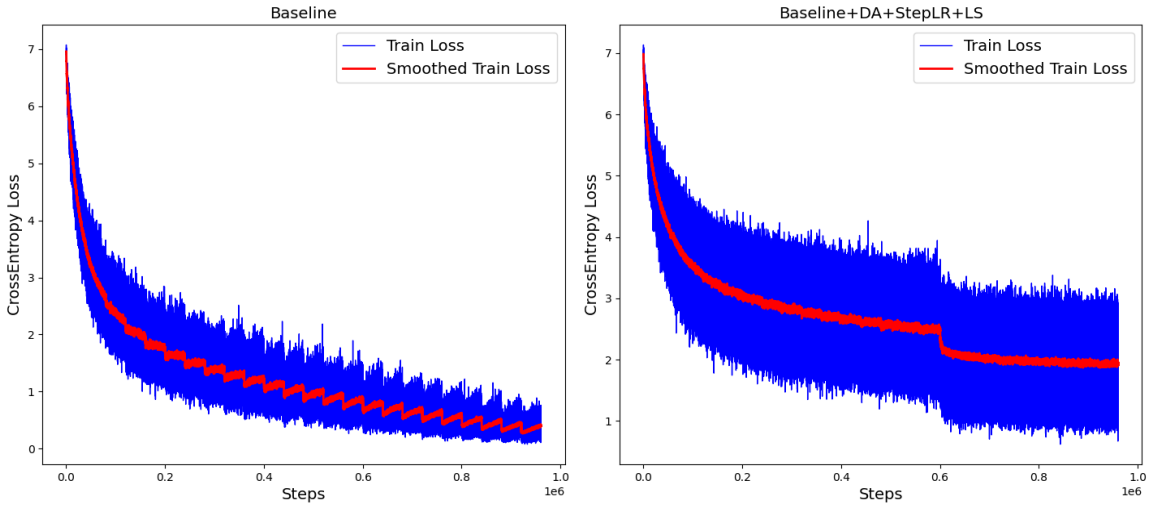

图 23 ResNet50 训练 Loss 曲线

左图为直接进行训练的结果，右图为增加数据增强的结果

蓝色曲线为原始各个 Batch 的 Loss，波动较大；

红色曲线是对一定区间做平滑处理的结果，可以反应 Loss 下降的整体趋势。

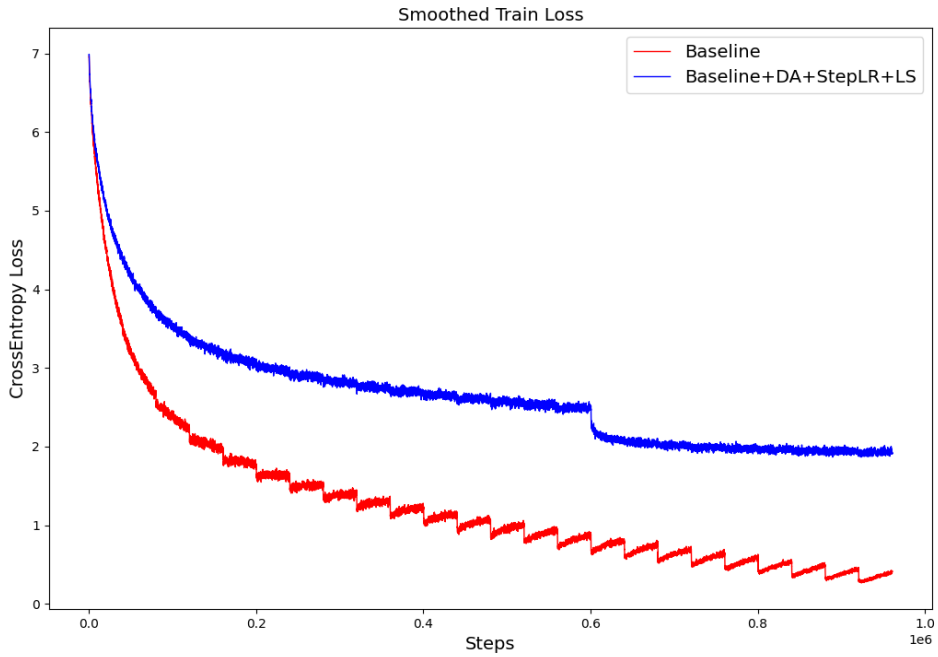

图 24 ResNet50 训练 Loss 曲线比较图

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

3.2.2 训练准确率与测试准确率

训练集和验证集准确率曲线如图 25，26 所示，相关性质表述同 3.1.2 中内容。

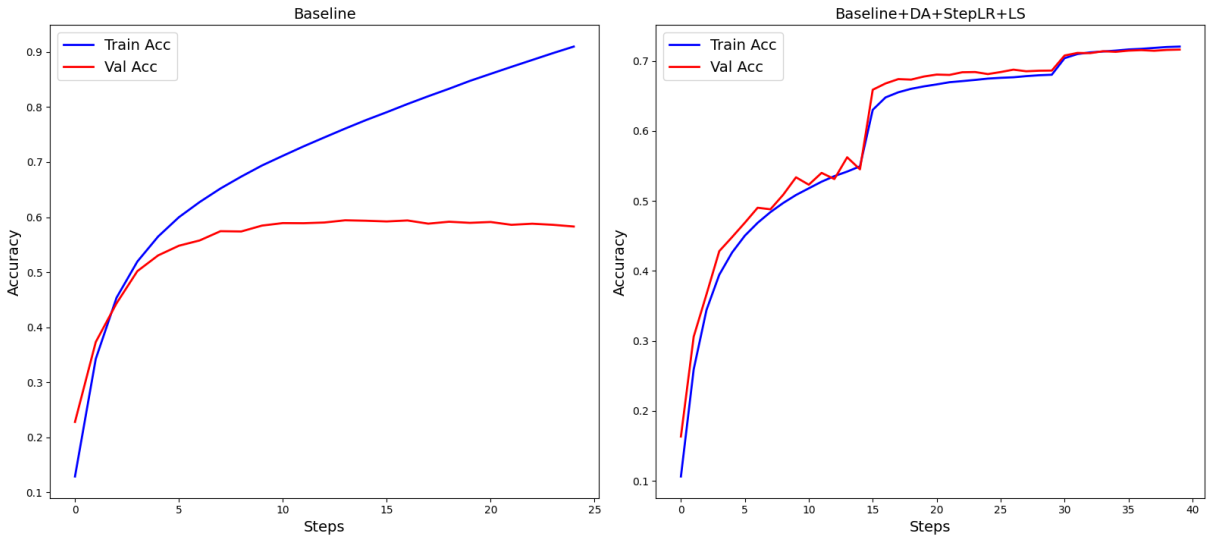

图 25 ResNet50 的训练集和测试集准确率

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

蓝色为训练集准确率，红色为验证集准确率

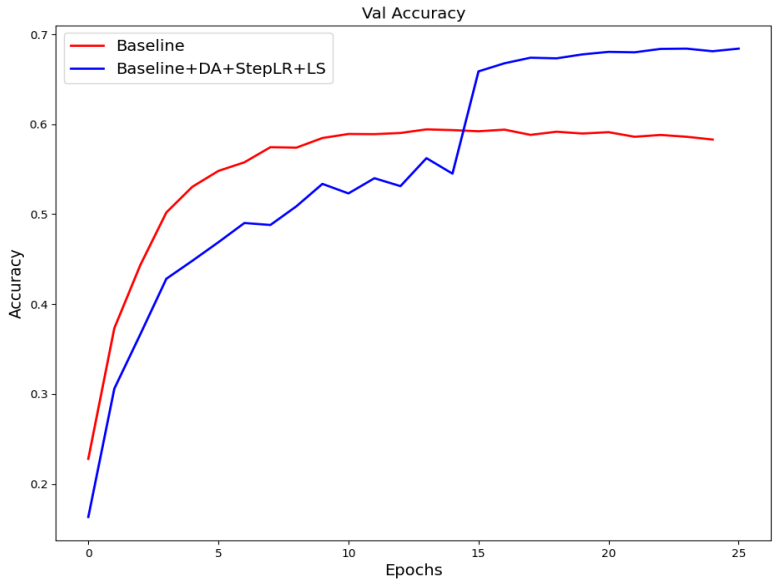

图 26 ResNet50 的测试集准确率比较（25Epoch）

Baseline 为基本对照，DA 为数据增强，StepLR 为阶梯式改变学习率，LS 为标签平滑

3.2.3 结果分析

我们的实验结果如表 4 所示，其中同样添加了 PyTorch 提供的预训练参数测试的结果。

表 4 的结果说明我们的实现具有正确性，在误差允许范围内可以说明我们的实现的正确性。而我们实现的 ResNet50 的训练速度远比 PyTorch 的慢，这可以理解为 50 层结构相比 18 层结构显著增加了残差连接和激活层的数目，而我们的实现为了内存

上的形式统一和正确性，宁可去多分配内存、多拷贝，而部分操作是完全可以做到无计算开销而保证正确性的，这也是我们需要改进的地方。其他表述同 3.1.3 节中的内容。

表 4 ResNet50 实验结果

方法	验证集准确率 Val Acc	单 Epoch 训练用时（秒）	单 Epoch 验证用时（秒）
Baseline	59.43%	3660	64
Baseline+DA+StepLR+LS	71.62%	3664	64
PyTorch	72.62%	3075	55

第 4 章 调试问题

在我们的实现过程中，不是所有改动都能只改动一处然后确定正确性，很多情况下牵一发而动全身，同时模型的修改如果出现错误可能不会导致结果的明显错误，但累积下来将影响最终正确性。在整个项目过程中调试的时间占到了整体开发时间的一半以上，我们调试用到的虽然都是“土方法”，但能远比观察代码更有效地发现问题，所以我们认为有必要对此做一些小结。此部分代码不在仓库中出现，但为调试带来了很大便利，故只在此处描述方法。

4.1 计算正确性与统计量验证

NVCC 有自带的调试工具 cuda-gdb，但是这个调试工具更适合调试核函数，在我们的使用过程中发现其单步执行速度过慢，效率太低，且无法直接在主机端获取显存内容，故我们都采用比较调试或者输出调试的方法。

由于在经过若干改动后，我们有时候需要校验输入输出是否如同期望在 CUDA 设备上进行计算，所以我们用循环实现了全连接层的矩阵乘法，卷积层的卷积操作，将 CPU 计算结果与 CUDA 结果进行逐一比对。在这种操作下我们排查出了**线性层权重参数矩阵与其形状描述不符合**的情况，**cudnnAddTensor 函数的 alpha 和 beta 分别为两个被加数的权重而非历史值的累积**，**cublasSgemm 进行矩阵读取时采用列优先下标而非 C++内存中普遍的行优先下标**，等，同时进一步确认了在卷积和矩阵乘法过程中没有出现如内存溢出、错误清零等导致的计算错误。

在排查问题的过程中，我们发现由于权重的参数量过大，样例代码中的调试代码并不适用于我们的调试，而快速排查问题的方式为**计算特征的统计量，即均值和方差**，统计量可以是某一层输出的整体统计量，通过观察整体统计量可以快速排查某一层是否出现问题，如方差达到 10 则说明在这一步的操作中明显出现了问题；也可以分别计算各个通道的统计量，如果只有某些通道异常，则可能说明输入的排列有问题，或者矩阵计算的维数与期望不同。通过整体统计量排查，我们发现了**初始化不正确的问题**，**全连接层改写矩阵乘法后只对某一列起作用的问题**，以及**输入数据 HWC 转 CHW 后颜色通道信息丢失变为灰度图的问题**。

4.2 整体正确性验证

无论怎样对局部问题进行排查，最有效的方式仍然是让算法完成整个训练-预测过程并查看其结果是否符合预期。对于我们的各类模型，我们实现了与其完全等价的 PyTorch 模型{py_src/main.py}，并将我们的结果与之对比，查看是否达到理想值。

我们选取 ImageNet 中类 ID 模 100 为 0 的 10 类作为小的“验证集”（下称 10 分类数据集），仅用此 10 类完成训练和验证，每个 Epoch 的训练仅需要 100 秒左右。我们在 PyTorch 中使用 ResNet18 在 10 分类数据集上不使用数据增强的方法达到了最高 84.4%的验证精度，而在模型调试的过程中，我们先后达到了 70%，78%，81%，82%，84%，87%的精度，其中 87%采用了数据增强。值得一提的时，即使我们的精度达到了 82%，在测试完整数据集时精度仅有 35%，与标准的 65%相去甚远，说明

通过**小数据集**对正确性进行验证需要关心**高精度的准确率**，这也说明了我们使用大数据集验证算法的可靠性。

调试期间，我们遇到了连续两天毫无进展的情况，模型参数过多，统计量和计算正确性验证已经完全无误，而具体参数值难以看出什么头绪，这时我们选择了一个极端而有效的方法：将模型与 PyTorch 进行**对拍**，即让我们的计算与 PyTorch 完全同步。为实现这一目的，我们首先将模型的所有参数保存为 NumPy 格式，并加载进 PyTorch 模型（参数保存参考{src/network.cu:Network::Forward}的第一段注释，参数加载参考{py_src/same.py:load_my_weight}），此外我们还需要生成一个固定随机排列供加载数据使用。对拍调试中我们逐层比较各个中间变量的输出值是否完全一样，权重梯度是否完全一样，以及中间变量梯度是否完全一样，如果不一样我们再去按照 4.1 中的方法做进一步调试。对拍主要帮助我们发现了**输入数据 HWC 转 CHW 后颜色通道信息丢失变为灰度图的问题**，**全连接层改写矩阵乘法后只对某一列起作用的问题**，以及最重要的**分裂节点清零 output 导致下一层的输入数据也清零并进一步导致梯度计算错误的问题**。分裂节点是我们实现残差连接的一个核心，该清零原理以及解决方案可以参考 2.3.3 节中的表述，在解决这一问题后，我们的模型基本不再出现问题，且连续若干步的 loss 值与 PyTorch 模型完全相同，应当可以认为实现上没有出现问题。

第 5 章 改进方向

5.1 In-Place ReLU 优化

ReLU 激活函数可以表示为：

$$y = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}$$

其导数值可以表示为：

$$\frac{\partial J}{\partial x} = \begin{cases} \frac{\partial J}{\partial y}, x > 0 \\ 0, x \leq 0 \end{cases}$$

由这两个式子我们可以发现，如果 ReLU 在$x = 0$时取导数值为 0（而不是同样在边界上的 1），则我们可以省去 ReLU 计算的输出空间，即不需要在内存中额外分配空间保存y，可以直接复用x的空间，这一操作被成为 in-place 优化，由于 ReLU 层频繁出现，做这项优化可以有效减少显存占用。对于 BatchSize 为 256 的 ResNet18，不使用 in-place 优化显存占用为 17221MB，而使用 in-place 优化占用为 14965MB。我们实现了部分 in-place 优化的操作，但是不具有正确性，时间紧张我们决定不在此处花费更多时间调试，留作日后优化。此处的显存占用也是先前章节提到的可以做显存优化的一个重要的点。

5.2 多分支的分裂节点和合并节点

我们实现的分裂合并节点都是二输入一输出或一输入而输出的，对于有n个分支或者求和的节点，用我们的实现相当于要开出$n - 1$块相同大小的空间用来保存中间变量，这显然是不必要的，我们可以使用一个 std::vector 来维护多分支的信息，从而做到对任意分支或合并都只开出一块额外空间。鉴于我们的模型没有多分支的情况，我们不再做进一步的优化。此外，我们的分裂节点实现需要进行会阻塞计算的拷贝，造成额外计算开销，理应可以通过设计避免这一点。

5.3 框架优化

我们的代码基本继承样例代码的架构，而这一架构面向对象与面向过程混用程度较高，对于同一个节点的所有功能可能会在多出实现，如特征维数确定、空间分配、计算、权重初始化等，当项目进一步增大时这不利于整体的维护，如果后续要做进一步开发应当将分散在各处的代码做进一步整合，从而提升可读性和可维护性。

对于计算图部分，我们显示地使用 AddEdge 函数进行建图，这种方式对熟悉框架的人不成问题，但不利于推广使用，试想如果有一条边没有连接可能导致整个模型的部分梯度不连续，从而产生隐形错误（不会抛出异常，只有结果不好）。我们可以通过重载运算符如加减乘除，以及对 Layer 地址精细化管理的方式进行隐形建图，这样我们可以做到像 PyTorch 一样的所见即所得以及自动求导功能，这样更有利于新模型的开发与调试。

第 6 章 总结

我们通过本次项目收获了许多内容，通过对样例代码的阅读与不断改进，我们相当于掌握了开发深度学习框架的基础，了解了如何用 C++去自己实现整个深度学习的流程。在这一过程中，我们对以下方面有了更进一步的认知：

- 为了调试算法我们了解了 im2col 以及矩阵乘法 stride 转变的技巧，推导了神经网络各层输入与输出的关系，输出梯度、权重、输入值与输入梯度、权重梯度的关系；
- 对各层或节点的内存结构与意义有了进一步的认知，了解深度学习中显存占用的具体分布及作用；
- 对优化器的内部实现有了更底层的认识；
- 对网络权重初始化有了细致的调试经验，权重初始化和 BatchNorm 参数初始化在各类框架中一般都在其类构造时自动进行，我们先前对这一点关注甚少，但实际操作时发现初始化对模型性能有较大影响；
- 对 CuDNN 以及 CuBLAS 的函数调用以及各类参数的含义有了充分的了解；
- 对 CUDA 核函数的调用有了一定认知（虽然主要用于赋值）；

在这一过程中我们也充分认识到现有的深度学习框架所提供的便捷性、灵活性和低内存开销性的确是冰冻三尺非一日之寒，是经过了很多优秀的软件工程师和算法工程师的优化才做到如今这一步。我们在项目中也认识到实现上的一些不足，比如没有通过核函数实现卷积和矩阵乘法并进行性能调优，没有选择更小一些的大型数据集来进行开发调试，在模型改动时总是为了完整性一下跨几步从而使调试变得及其困难等，这些也是我们应当日后进一步努力或戏曲教训的地方。

参考文献

[1] He, K. , et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) IEEE, 2016.

[2] Papakipos, Zoë, et al. "Results and findings of the 2021 Image Similarity Challenge." arXiv preprint arXiv:2202.04007 (2022).

[3] Wang, Wenhao, Weipu Zhang, et al. "Bag of tricks and a strong baseline for image copy detection." arXiv preprint arXiv:2111.08004 (2021).

[4] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009.

[5] https://pytorch.org/vision/stable/_modules/torchvision/models/resnet.html#resnet50

[6] https://raw.githubusercontent.com/soumith/imagenetloader.torch/master/valprep.sh

[7] https://gitlab.bitigca.cool/zzzc18/resnet_cudnn

[8] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. PMLR, 2015.

[9] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html?highlight=batchnorm#torch.nn.BatchNorm2d

[10] https://docs.nvidia.com/deeplearning/cudnn/pdf/cuDNN-API.pdf

[11] https://github.com/hongchu098/opencv_torchvision_transforms

[12] https://pytorch.org/docs/stable/generated/torch.optim.SGD.html?highlight=sgd#torch.optim.SGD

[13] Müller, Rafael, Simon Kornblith, and Geoffrey E. Hinton. "When does label smoothing help?." Advances in neural information processing systems 32 (2019).

[14] https://github.com/rogersce/cnpy

[15] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

[16] Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." arXiv preprint arXiv:1706.02677 (2017).