课程报告

刘天毅 2021310655

一、 所选题目

选题 2：卷积神经网络运行框架

二、 使用到的库

1. blas 与 cblas 库

线性代数运算库及其 C 语言接口。本次大作业中用到了该库的矩阵与向量乘法功能。网址：https://netlib.org/blas。

2. CImg 库

一款轻量级、开源图像处理库，安装与使用非常方便。本次大作业中仅使用到该库的图像显示功能。网址：http://www.cimg.eu/index.html。

三、 实现思路

1. 基本数据格式

神经网络中的基本数据格式为张量（tensor），常用的张量维度从 1 维到 4 维不等，比如特征向量（1 维）、灰度图像（2 维，高度 H×宽度 W）、彩色图像与特征图（3 维，通道数 C×高度 H×宽度 W）、卷积核（4 维，输出通道 Co×输入通道数 Ci×高度 H×宽度 W）。

在 C++环境下，需要用多维数组来处理高维数据，而这涉及到指针的操作。通常来说，我们希望指针的级数不要超过两级，更多的指针级数会使得数据的处理更加复杂和困难。因此，在此次大作业的设计中，仅会使用到两级指针来实现 1 维至 4 维的张量数据储存。具体的实现方式如下（示意图见图 1）：

- **1d 格式**：使用指针创建一维动态数组，用来储存 1 维的特征向量与 2 维的灰度图像。对于 2 维张量，按照先行后列的顺序展开保存在一维数组中。
- **2d 格式**：使用二级指针创建指针数组，其中每个指针都指向一个一维的动态数组，用来储存 3 维的特征图与 4 维的卷积核数据。对于 3 维张量（C×H×W），首先按照通道 C 进行展开，得到 C 个 2 维张量，然后按照先行后列的顺序将每个通道的数据保存在相应的一维数组中；对于 4 维张量（Co×Ci×H×W），首先按照输出通道 Co 进行展开，得到 Co 个 3 维张量，然后按照 W→H→Ci 的顺序将每个 3 维张量展开为 1 维数据，并保存在相应的一维数组中。

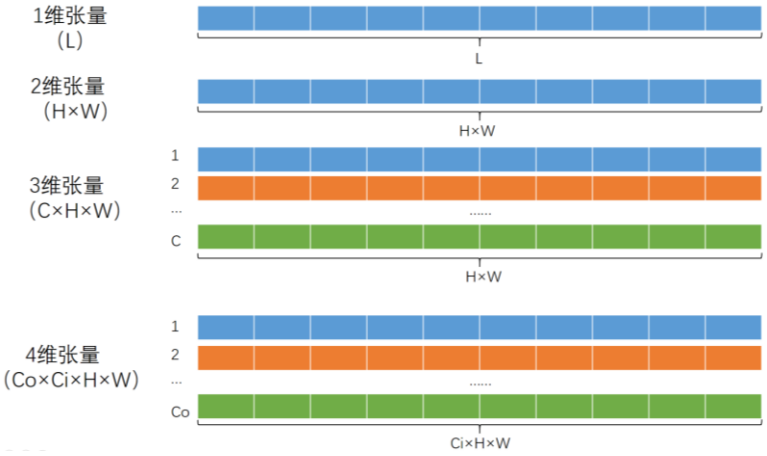

图 1　C++储存高维张量方式示意图

按照上述方式储存的数据与 pytorch 框架的张量操作顺序相匹配，便于后续模型的读取与数据的交互。

2. 全连接层

全连接层为神经网络中最基础的结构，其数学表达式为

$$y = Wx + b$$

其中 W、b 为全连接层的参数。

本次大作业定义了全连接层类 fully_connected，实现了参数的存储、预训练参数的导入与前向传播功能。其实现方式描述如下：

- 构造函数 fully_connected

构造全连接层时，需要提供输入与输出的维度信息，从而为参数 W、b 开辟相应的存储空间。为了适配 blas 数值计算库的接口定义，权重参数 W 与 b 采用 1d 格式进行存储。

另外，全连接层中包含有一个标记信号 valid，用于指示当前全连接层的参数是否已完成初始化。在构造函数中，会将 valid 首先置为 false，表明此时的权重参数无效，无法调用前向传播函数。

- 参数导入函数 set_weight

导入预训练的参数 W、b，并将标记位 valid 置为 true，表明全连接层已完成初始化，可以进行前向传播。

- 前向传播函数 forward

在前向传播函数中，首先判断 valid 状态，若为 false 则给出错误信息并退出。前向传播的计算过程较为简单，直接调用 blas 库的矩阵向量乘法函数 cblas_sgemv()即可。注意 blas 库提供的函数接口具有较高的泛用性，因而输入参数十分庞杂，需要仔细阅读说明文档，理解清楚各个输入参数的含义。

3. 池化层

目前在神经网络中的池化操作大多使用最大池化方法，因此本次大作业将最大池化层作为设计目标。

以 kernel size=2, stride=2 为例，最大池化的操作示意图如下

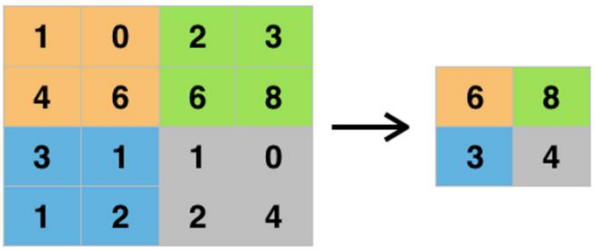

图 2　2×2 最大池化示意图

其严格的数学表达公式为

$$\text{output}(C_k, i, j) = \max_{i*S \le m < i*S+K} \max_{j*S \le n < j*S+K} \text{input}(C_k, m, n)$$

其中C_k为输入通道编号，K 为核尺寸 kernel size，S 为步长 stride。

本次大作业定义了最大池化层类 max_pooling，实现了最大池化层的前向计算功能，其实现方式描述如下：

- 构造函数 max_pooling

构造最大池化层时，需要提供核尺寸 kernel size 与步长 stride。

- 前向传播函数 forward

前向传播函数可依据上述的数学公式进行计算，输入数据为 2d 格式保存的 3 维张量（具体的存储方式如前文所述），因而需要按照存储方式对下标值进行一定的换算。

4. 激活函数

深度学习中有许多类型的激活函数，包括 sigmoid 函数、tanh 函数、ReLU 函数等。由于 ReLU 函数计算方便，并且能够有效避免梯度消失与梯度爆炸问题，因此在大规模神经网络中得到了广泛的应用，本次大作业选择 ReLU 函数作为激活函数进行实现。

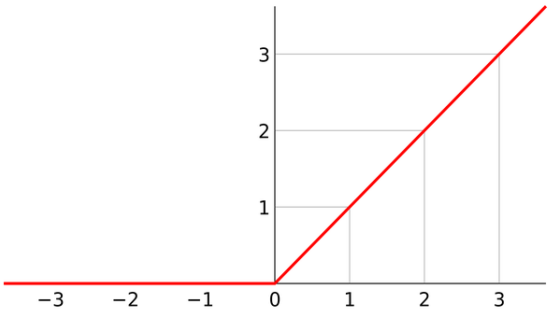

图 3　ReLU 激活函数

在神经网络中，激活函数的输入数据可能是全连接层、卷积层或者池化层的输出数据，它们的数据格式并不相同。因此，本次大作业在实现 ReLU 激活函数时，定义了**两个重载函数**，分别用来处理一开始定义的 1d 与 2d 格式的输入数据。另外，类比于 pytorch 框架中的 inplace 功能，我设计的 ReLU 激活函数也具有类似的选项，可以实现**原地计算**。

5. 卷积层

卷积层的实现是本次大作业的核心环节，其数学原理如下

$$\mathrm{out}(\mathrm{Co_k}) = \mathrm{b}(Co_k) + \sum_{\substack{i,j=1,2,\dots,K \\ n=1,2,\dots,C_i}} W(Co_k, n, i, j) * \mathrm{in}\left(n, h_i, w_j\right)$$

其中Co_k为输出通道编号，K 为卷积核尺寸，C_i为输入通道数。

注意到本次大作业需要实现的网络模型包括 AlexNet、ResNet 与 VGGNet，因此需要卷积层可以灵活调整不同的卷积核尺寸、步长与填充（padding）。

本次大作业定义了卷积层类 convolution，实现了卷积层的各种配置、预训练参数的导入与前向计算的功能，其实现方式描述如下：

- 构造函数 convolution

构造卷积层时，需要提供输入与输出的通道数、kernel size、stride 与 padding，从而为参数 W、b 开辟相应的存储空间，并确定卷积的运算方式。

类似地，卷积层中也包含有一个标记信号 valid，用于指示当前卷积层的参数是否已完成初始化。在构造函数中，会将 valid 首先置为 false，表明此时的权重参数无效，无法调用前向传播函数。

- 参数导入函数 set_weight

导入预训练的参数 W、b，并将标记位 valid 置为 true，表明卷积层已完成初始化，可以进行前向传播。

- 前向传播函数 forward

在前向传播函数中，首先判断 valid 状态，若为 false 则给出错误信息并退出。接下来会判断是否需要 padding，若需要 padding，则会对输入特征图进行 zero padding 操作。在计算卷积时，采用的方法是多次调用 blas 库的向量内积函数 cblas_sdot()，实现乘累加操作。

6. 残差模块

ResNet 网络的基本结构为残差模块（residual block），如下图所示。本次大作业也对残差模块进行了建模与封装。

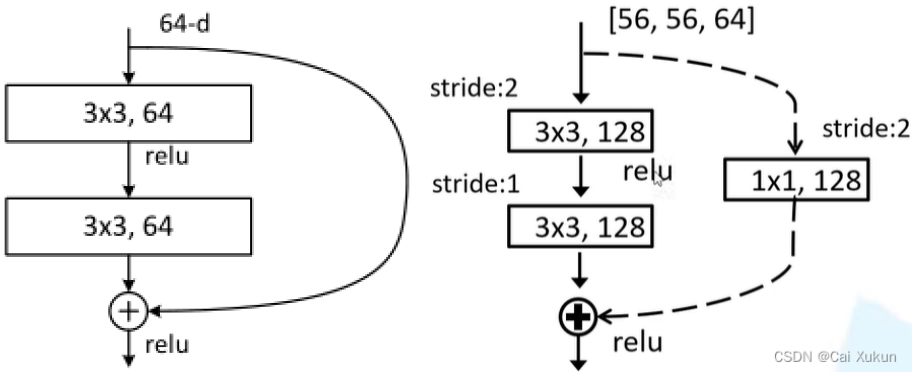

图 4　residual block 示意图。左：特征图尺寸一致，右：特征图尺寸改变

通过阅读文献[1]以及网络上一些 ResNet 的 pytorch 实现代码，可以得出 residual block 的结构主要有两种：1）通道数与特征图尺寸均不变（上图左），此时直连通路可以直接与卷积层的输出结果相加；2）通道数增加、特征图尺寸减半（上图右），此时直连通路与卷积层输出结果的尺寸不一致，需要经过一定的变换才能相加。

本次大作业实现的残差类综合考虑了上述两种类型，可通过不同的初始配置实现这两种类型的残差模块。其中结构 2）的实现参考了一些 pytorch 代码的实现方式，采用了 1x1 卷积对直连通路进行处理，以匹配最终的输出尺寸。

本次大作业定义了残差模块类 residual，实现了残差模块的不同配置、预训练参数的导入与前向计算的功能，其实现方式描述如下：

- 构造函数 residual

构造残差模块时，需要提供输入与输出的通道数，以及 stride，将根据这些信息初始化两层卷积层。若输入通道数≠输出通道数，或者 stride≠1，则还需初始化一个 1x1 卷积层用于直连通路的变换。

- 参数导入函数 set_weight

分别调用各卷积层的 set_weight 函数，完成预训练参数的导入。

- 前向传播函数 forward

按照上图所示的数据流，依次调用相应的 forward 函数即可。注意判断 residual block 的特征图尺寸是否发生改变，并按照相应的数据流执行。

四、 模型结构

根据 MNIST 数据集的格式，以及 AlexNet、ResNet 与 VGGNet 网络的特点，本次大作业针对 MNIST 设计了三种模型的简化版本，用于验证代码的功能。各模型的结构如图 5 所示，其中卷积层的符号含义为 Conv(in_channel, out_channel, kernel_size, stride, padding)，全

连接层的符号含义为 Linear(in_size, out_size)：

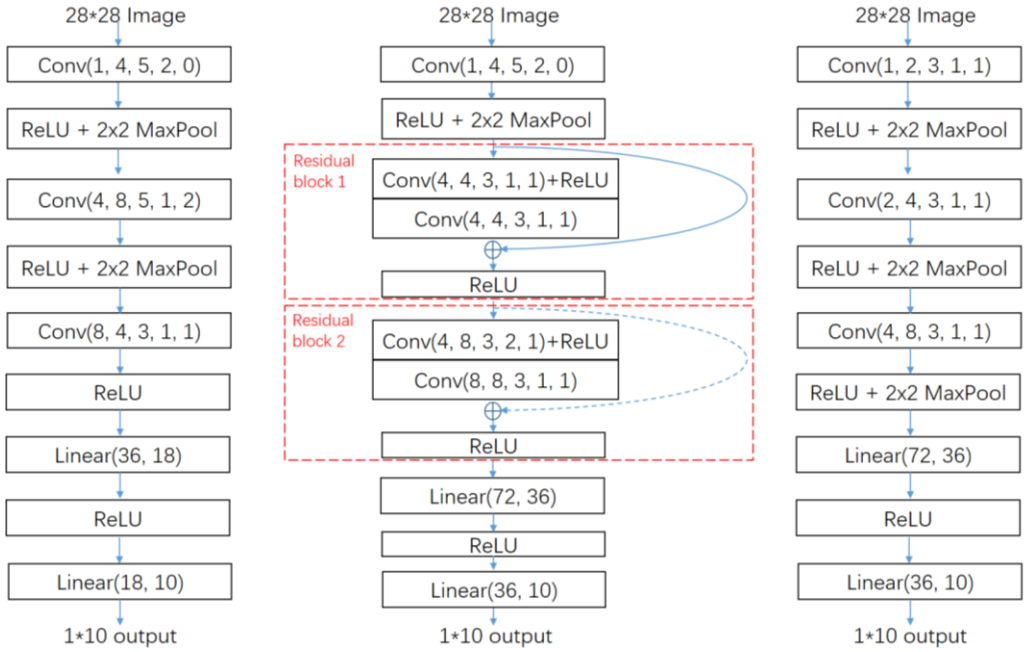

图 5　针对 MNIST 数据集设计的 AlexNet（左）、ResNet（中）与 VGGNet（右）

注意到每种模型的简化版本都全面覆盖到了原始模型可能出现的参数配置与连接模式，因此使用这些简化模型的实验结果，足以证明该框架能够实现 AlexNet、ResNet 与 VGGNet 网络的原始结构。

模型采用了文本文件的方式进行存储，按照网络层次由上到下，依次将每项参数保存成一行数据。在 Python 端编写了相应的脚本，可以将 pytorch 的模型文件按照此格式重新保存。在 C++端，每种网络都具有读取模型文件的函数，可以将 pytorch 训练后的网络参数导入进来。

五、 结果展示

使用 pytorch 对 AlexNet、ResNet 与 VGGNet 网络训练 10 epoch 后，模型在 MNIST 测试集上的准确率均达到 97%以上。将此时的 pytorch 模型参数导入到 C++模型中，利用 MNIST 测试集数据验证 C++模型的功能是否正确。

使用 CImg 库展示输入图片，并在标题位置显示图片的 label 与网络输出的预测结果，部分示例如图 6 所示：

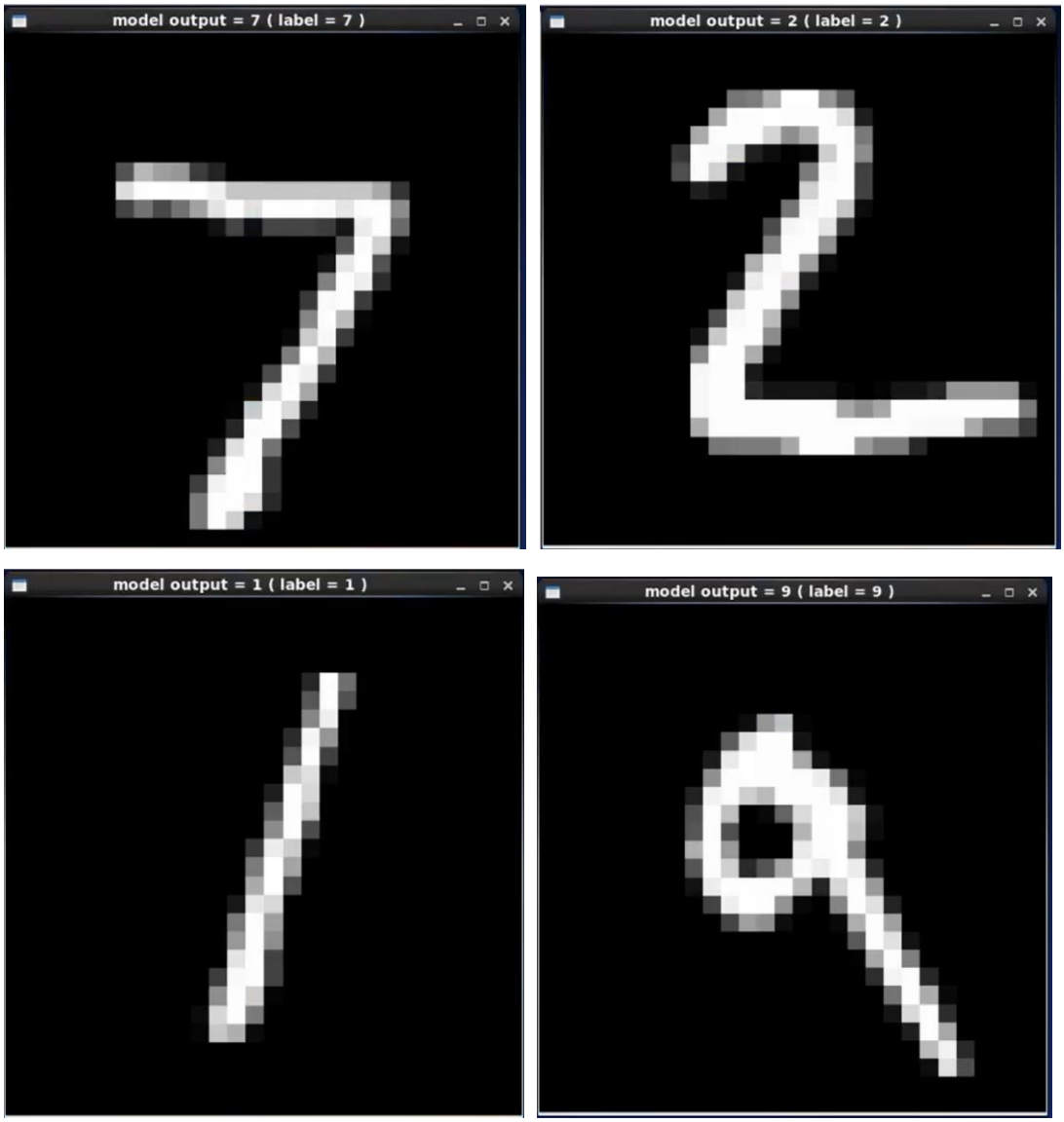

图 6　C++卷积神经网络框架的输出结果

经实验验证，AlexNet、ResNet 与 VGGNet 三类模型均能够输出正确的预测结果，表明模型的功能正常，实现了预期的设计目标。

六、 总结

本次大作业，我使用 C++语言，完成了卷积神经网络运行框架的编写，可以支持 AlexNet、ResNet 与 VGGNet 的前向推理与结果展示功能，并在手写数字数据集 MNIST 上进行了实验验证。框架具有一定的通用性与灵活性，可以自由配置通道数、卷积核尺寸、步长、padding 等参数，实现不同的网络结构。受时间所限，未能完成神经网络的反向传播功能。

通过本学期的课程学习与作业实操，我了解了 Unix 环境的文件系统、内存管理与进程间通信的相关知识，熟悉了在 Unix 环境下的 C 语言编程的基本思路，学会了 gcc、Git、Makefile 等工具的基本用法，这对于我后续的科研工作有着很大的帮助。最后，感谢老师与助教一学期以来的付出。

附录

1. 提交文件说明

./MNIST：MNIST 手写数据集

./models：按照文本文件格式保存的预训练模型

- ├ model_AlexNet
- ├ model_ResNet
- └ model_VGGNet

./python：相关的 Python 代码与模型

- ├ load_mnist.py：导入 MNIST 数据集，生成 pytorch 格式的 Dataset
- ├ model_AlexNet.py：AlexNet 网络的定义文件
- ├ model_ResNet.py：ResNet 网络的定义文件
- ├ model_VGGNet.py：VGGNet 网络的定义文件
- ├ model_save.py：将 pytorch 模型文件保存为文本文件格式，供 C++读取
- ├ train_model.py：训练 pytorch 模型的代码
- └ *.pth：训练好的 pytorch 模型文件

./src：C++卷积神经网络运行框架的代码文件

- ├ basic_components.h, basic_components.cpp：基础模块的定义与实现
- ├ cblas.h：cblas 库的接口定义
- ├ CImg.h：CImg 图像库文件
- ├ libblas.a, libcblas.a：blas 与 cblas 链接库文件
- ├ main.cpp：主函数，实现网络推理与结果展示
- ├ Makefile 文件
- └ Models.h, Models.cpp：AlexNet、ResNet 与 VGGNet 网络的定义与实现

2. 代码运行说明

在./src 路径下：

- make main：编译工程
- make do_Alexnet：导入 AlexNet 模型参数，进行网络推理与结果展示
- make do_Resnet：导入 ResNet 模型参数，进行网络推理与结果展示
- make do_VGGnet：导入 VGGNet 模型参数，进行网络推理与结果展示
- make clean_main：清除编译的文件

参考文献

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition, 770–78, 2016.