

本科毕业设计



设计题目：面向深度学习算法的异构加速计算技术研究

姓 名： 张朕银

学 号： 16300720049

院 系： 微电子学院

专 业： 微电子科学与工程

指导教师： 陈更生 职 称： 高级工程师

单 位： 微电子学院

完成日期： 2020 年 6月 2日

# 摘要

近年来计算需求爆炸式增长，人们提出异构计算的概念，意图通过多种架构的设备之间协调工作的方式，进行计算任务的分配调度，进而实现计算速度的提高。FPGA作为一种具有高度并行潜力的器件，在异构计算领域崭露头角。而深度学习算法在成为人工智能领域的热点的同时，也带来了巨量的计算需求。在诸多深度学习算法中，卷积神经网络又是热门中的热门。LeNet网络结构是卷积神经网络的经典架构，具有体量小且精度高的特点，面对手写文字识别等任务有着较好的效果。

本文意图在CPU、GPU和FPGA上分别部署一个用于MNIST手写识别的LeNet实例，以观察GPU和FPGA在神经网络方面的加速效果。另外，本文还设计了一个前端页面，用于三个架构的实例的运行状态远程监测，并附有图片在线识别功能。本文采用的LeNet实例在CPU上的识别精度可达97.8%，在GPU上的识别精度可达98.0%，设计的LeNet核在FPGA上的识别精度可达96.7%。

**关键词**：深度学习，异构计算，FPGA，LeNet，MNIST手写数字识别

# ABSTRACT

In recent years, the demand for computing has been exploding. People have proposed the concept of heterogeneous computing, and intend to allocate and schedule computing tasks by coordinating work among devices with multiple architectures, thereby increasing the speed of computing. As a device with high parallel potential, FPGA has emerged in the field of heterogeneous computing. While deep learning algorithms have become a hotspot in the field of artificial intelligence, it has also brought huge computing needs. In many deep learning algorithms, convolutional neural networks are the focus of the focus. The LeNet network structure is a classic architecture of convolutional neural networks, which has the characteristics of small volume and high accuracy and has good results in the task of handwritten text recognition.

This article intends to deploy a LeNet instance for MNIST handwriting recognition on CPU, GPU and FPGA respectively to observe the acceleration effect of GPU and FPGA in neural network. In addition, this paper also designs a front-end page for remote monitoring of the running status of the three architecture examples, with an online image recognition function. In this paper, the recognition accuracy of the LeNet instance on the CPU can reach 97.8%, the recognition accuracy on the GPU can reach 98.0%, and the recognition accuracy of the designed LeNet core on FPGA can reach 96.7%.

**Keywords**: deep learning, heterogeneous computing, FPGA, LeNet, MNIST handwritten digit recognition

目录

[摘要 2](#_Toc10608)

[ABSTRACT 2](#_Toc1660)

[第一章 绪论 4](#_Toc17049)

[1.1 研究背景及意义 4](#_Toc16625)

[1.2 研究现状 5](#_Toc26766)

[1.3 论文主要研究内容与创新点 5](#_Toc6589)

[第二章 基础知识与相关背景 5](#_Toc7755)

[2.1 深度学习 5](#_Toc22459)

[2.1.1 DNN 6](#_Toc26578)

[2.1.2 CNN 7](#_Toc3402)

[2.2 异构计算 7](#_Toc15064)

[2.2.1 基本原理 7](#_Toc15036)

[2.2.2 CPU与GPU 8](#_Toc14869)

[2.2.3 FPGA 9](#_Toc13549)

[第三章 LeNet在CPU/GPU/FPGA上的分别部署 11](#_Toc14318)

[3.1 平台、语言及工具链 11](#_Toc31165)

[3.1.1 开发语言 11](#_Toc9102)

[3.1.2 PyTorch深度学习框架 11](#_Toc10572)

[3.1.3 Vitis硬件部署平台 12](#_Toc23469)

[3.1 基于DNN的MNIST识别程序 12](#_Toc21032)

[3.1.1 DNN网络设计与训练 12](#_Toc31886)

[3.1.2 网络的修改 13](#_Toc28617)

[3.2 使用PyTorch在CPU/GPU上部署LeNet 14](#_Toc11009)

[3.2.1 LeNet网络结构 14](#_Toc13839)

[3.2.2 代码编写 15](#_Toc11374)

[3.3 使用VITIS在FPGA上部署LeNet 16](#_Toc22950)

[3.3.1 LeNet网络参数的获取 16](#_Toc23600)

[3.3.2 网络结构的设计 17](#_Toc17520)

[3.3.3 网络结构的验证 19](#_Toc20678)

[第四章 面向LeNet实例部署的前端设计 19](#_Toc24173)

[4.1 总框架结构及其实现 19](#_Toc22447)

[4.2 监测页面的设计与实现 20](#_Toc5816)

[4.2.1 前后端响应 20](#_Toc25824)

[4.2.2 进程间通信 21](#_Toc30095)

[4.2.3 各版本LeNet的代码调整 22](#_Toc15162)

[4.2.4 页面崩溃问题的解决 23](#_Toc32109)

[4.2.5 图像刷新问题的解决 23](#_Toc13131)

[第五章 测试与前端效果 24](#_Toc4899)

[5.1 前端效果一览 24](#_Toc22192)

[5.2 CPU/GPU/FPGA版本LeNet的识别率 27](#_Toc27766)

[第六章 工作总结和展望 28](#_Toc11473)

[6.1 工作总结 28](#_Toc15851)

[6.2 后续工作展望 28](#_Toc20107)

[参考文献 29](#_Toc24981)

[致谢 30](#_Toc26536)

# 

# 第一章 绪论

## 1.1 研究背景及意义

近年来计算需求爆炸式增长，人们提出异构计算平台，意图填补对速度和能效的迫切需求。异构计算平台本质特点有二，即高并行度和异构性。

高并行度应用在计算任务之间相互独立的情况，相比使用CPU计算而言，可极大提高效率。GPU和FPGA的天然高并行性因此得到发挥。

异构性指异构平台下有多个架构相异的子系统，平台主机需要根据子系统的结构特点，为其调度不尽相同的计算任务。

相比于GPU和CPU，FPGA作为异构加速平台的组分有以下优势：

(1)功耗低。面对同样的计算任务，FPGA的功耗比ASIC稍高，相比CPU和GPU低得多。

(2)延时低。相比CPU，FPGA拥有很高的并行度。相比于GPU，FPGA拥有足够高的并行度，同时流水线深度不受限制，而GPU的流水线深度受体系结构的限制。因而使用FPGA进行计算可以获得非常低的延时。

(3)灵活性高。由于FPGA可重构的体系结构特点，它接受反复配置。基于这一点，可实现FPGA动态局部实时重构。值得注意的一点是，FPGA的重构花费的时间通常在微秒到毫秒量级，在频繁重构的情形下不容忽视。

另外，由于ASIC开发周期长，且可重构性较差，在异构计算平台中鲜有使用。

深度学习算法在计算机视觉、语义识别等领域均得到重大突破，成为研究热点。著名深度学习算法模型有DNN、CNN、RNN、GAN、RBF等，其特点为模型网络化、大量参数和高并行度。异构计算平台应对这样的应用场景具有针对性优势。

## 1.2 研究现状

FPGA异构加速计算早有先例。

学术上较为著名的两种异构计算模型分别为CUDA和OpenCL[1]。CUDA在2007年由NVIDIA推出，只支持NVIDIA GPU。而OpenCL在2008年由开源标准机构Khronos集团正式推出。之后，Altera于2010加入Khronos，为OpenCL支持FPGA的相关标准的制定作出了积极贡献。

此外，Xilinx公司也开发出SDAccel的开发环境，支持OpenCL、C/C++设计应用，利用FPGA进行加速计算，可获得高达25倍的性能功耗比提升。

传统的加速计算主要思路为增加并行与降低访存延时。增加并行的思路主要有多芯片并行、多核并行、多线程并行和指令并行；降低访存延时的思路主要为构建层次化存储系统，即寄存器、本地存储器、缓冲区、主存储器的层级架构[2]。异构计算的基本思路则是将GPU、FPGA等设备视作CPU的协处理器，并设计新的编程语言或在原有编程语言基础上修改，进行计算任务的调配。

将FPGA用于云计算的技术也日趋成熟[3]。一些FPGA云加速服务也已投入商用。腾讯云、微软云、阿里云、华为云等公司或部门陆续推出对FPGA云加速计算服务的支持。

## 1.3 论文主要研究内容与创新点

论文主要研究内容为，将LeNet在CPU、GPU、FPGA上分别部署，并观察它们之间的性能异同，并设计一组网页来实时查看部署实例的工作状态，以及对实例进程进行控制。主要的创新点在于，制作网页而非本地应用来查看部署实例状态，这对于工具链后续的上云计划至关重要。

# 第二章 基础知识与相关背景

## 2.1 深度学习

深度学习是目前人工智能领域中最主要也最热门的分支，而深度学习最主要的内容则为神经网络。神经网络的思想最早可追溯到1943年的一个名叫“MCP”的人工神经元模型，该模型主要模拟了神经元的多输入特性，以及输入输出关系的非线性特性。尔后于1958年，Rosenblatt发明了感知机（perceptron）算法，使用MCP模型来处理分类问题。但感知机算法的局限性在于，它只能处理线性分类问题，对于非线性分类问题（例如异或映射）则束手无策。

更之后，Hinton在1986年提出了一个基于感知机的算法：多层感知机（MLP）。该算法采用反向传播的参数优化方式，并且神经元的非线性映射采用sigmoid函数，成功越过了单层感知机对非线性分类问题的局限性。该模型在1989年被Robert Hecht-Nielsen通过数学方式证明为具有万能逼近的特点[4]。此时的多层感知机为后续的诸多神经网络（NN）模型打开了一扇大门。

### 2.1.1 DNN

DNN，Deep Neural Network，深度神经网络，即多层感知机。其基本结构是具有隐含层的神经网络拓扑结构，两层之间的神经元两两连接，形成全连接。单层感知机无法处理非线性分类问题，而多层感知机能够处理非线性分类问题，就是因为其中的隐含层加入了非线性因素。

如果说全连接层中的连接就是神经元的突触，那么每个神经元就在通过“突触”上的权重和sigmoid函数处理上一级传递而来的信息。一个神经元会将上一级传递来的信息分别乘上突触上的权重，然后通过一个sigmoid函数转交给下一级。正因此，多层感知机中神经元的行为可以被等效成为矩阵乘法，进而交由GPU、FPGA和专门设计的ASIC电路等并行度高的处理设备来处理，以加速其运行。此处的加速原理在后续的异构计算中会提到。

#### 2.1.1.1 梯度消失问题

DNN模型面临的一个效率危机便是梯度消失问题。由于Sigmoid函数自身的特性，当一个神经元收集到的权重×信号所得出的值偏离0较远时，对应位置的梯度会下降到非常低，而反向传播的优化算法又十分依赖这一梯度，因而此时数据的优化就会近乎停滞。这样的梯度消失效应在DNN网络层数较深的时候尤为严重，因为很可能存在一条神经元链，其中多个Sigmoid函数相串联，在总梯度的因子中形成了多个小于1的数的累乘链，从而使得最终的梯度非常小。为应对这类问题，ReLU函数被提出，有效地抑制了梯度消失问题，并保留了相当的非线性属性[5]。

#### 2.1.1.2 梯度爆炸问题

与梯度消失问题类似，一个DNN网络的初始化权重通常是随即给定，那么一旦网络中形成了一条神经元链，其中多个大于1的神经元梯度累乘，得到的最终梯度将远大于1，从而形成梯度爆炸的效果。ReLU函数的引入同样可以较好地解决这一问题。

#### 2.1.1.3 过拟合问题

在神经网络的应用中有一个重要指标是“泛化能力”。一个神经网络通常需要接受一定的“内容：标签”结构的训练数据库进行训练，而接受另一个测试数据库来进行测试。如果一个神经网络在训练数据库上训练太久，就会过分地拟合训练数据库，从而偏离测试数据库，此时网络在训练数据库上的精度比之前要更高，但在测试数据库上的精度会降低。

为解决过拟合问题，可以通过交叉检验的方式，例如k折交叉检验，即把初始采样切分为k份，其中k-1份用于训练，1份用于检验，并且将这一过程重复k次以确保每一份都曾被用户训练和检验，最终取k次重复的结果的平均。另外正则化方法也是解决过拟合问题的一个非常有效的方法，常用的分支有L\_1正则和L\_2正则。

#### 2.1.1.4 局部最优解问题

现有的优化算法通常基于梯度下降的算法，而梯度下降算法不可避免的一个问题就是局部最优解问题。如果一个任务不是“恒凸函数”的，那么规定任意初始值，并根据一定的梯度下降算法，必定有一定的可能性使得状态下降到一个无法再下降的极值。根据给定的梯度下降算法，该状态不可再被优化，但它并不是整个网络于全局的最优解，只是局部的最优解。

针对这一问题，dropout算法被Hinton提出。Dropout算法的基本思路是对于一个神经网络，施加一定的随即概率使之偏离当前状态，相当于“变异”概率。该方法能够充分防止网络状态掉入局部最优解。

#### 2.1.1.5 剪枝优化与量化压缩

一个隐含的网络拓扑优化思路是，如果某一个神经突触上的权重非常接近零，无法提供很多信号，那么这条突触就可以被剪掉，从而优化网络拓扑，减轻神经网络的运算负担，从而提高其部署在设备上的运行效率。这样的剪枝行为对网络的复杂度降低非常重要，对减轻网络的过拟合问题也有好处。

另外，原生的DNN中，参数和信号均采用双精度浮点数的格式，有时采用的精度比双精度浮点数还要更高，这有利于最初的模型搭建与验证，但不利于其在硬件上的部署。硬件上采用精度过高的浮点数进行运算会加重乘法运算的运算量，相应的专用集成电路或FPGA软核的电路规模也会迅速上升，电路的功耗也会一并增大。这使得高精度带来的价值有些得不偿失。因而，一个必须考虑的问题就是将DNN网络所采用的高精度浮点数进行量化压缩，通过降低其精度位数的方式缩减网络复杂度，同时不损失过多的测试准确性。

### 2.1.2 CNN

CNN，convolutional neural network，卷积神经网络。相比DNN而言主要引入了两种新结构，即卷积层和池化层。

卷积层使用一个或一组卷积核，对输入信号进行卷积操作。此处的卷积操作与数学上定义的卷积操作的遍历方向相反，但核心思想一致，因而学界默许这种称呼。卷积操作对很多信号都能够十分有效地提取特征，在图像识别、语音识别、自然语言处理领域均有应用。面对不同的信号，卷积核可以是一维、二维、三维的，也可以是一个点。

池化层依照步幅和范围两个参数，对输入信号进行降采样操作。该操作能够在不损失过多特征信息的情况下减少信息量，对于缩减网络参数规模而言非常重要。

## 2.2 异构计算

### 2.2.1 基本原理

异构计算是一种并行计算技术，是分布式计算的子集。异构计算技术通过对某一计算任务的合理拆分与调度，将这一计算任务划分为多个子任务，并将这些子任务分配给架构各异的计算单元来处理。基于这种特性，异构算法能够根据计算单元的计算能力特点进行任务分配，从而充分发挥它们的效能。

一般来说，根据编程模型，一个庞大的计算任务是由若干个更小的子任务组成的，这些子任务可能需要使用其它任务的输出，因而将会产生一定的依赖关系。并且，这些子任务本身的运算类型，例如整型运算、浮点运算、向量运算、矩阵运算、张量运算等，使得它们常常显现出具有特征的依赖或非依赖关系。很多时候子任务划分到这些有特征的层级便不再继续向下划分，虽然继续向下划分是可行的。比如一个任务T，需要按次序进行5组矩阵乘法，其中第i个矩阵乘法中两个乘数的规模分别为mi×ni、ni×li，且矩阵内的数据均考虑为浮点数，那么任务T就可以向下划分成为5个有着次序依赖关系的矩阵乘法任务。若不考虑读写任务，矩阵乘法任务还可以再向下细分为mi×li个相互独立的规模为ni2的向量乘法任务，而每个向量乘法任务又可以向下划分成为ni2个浮点乘法任务和ni-1个加法任务。在各自的领域内，浮点乘法任务两两之间相互独立，无需等待对方的运算结果便可以开始运算；在各自的领域内，向量乘法任务两两之间也相互独立。通常我们希望任务T划分到矩阵乘法便不再特意划分，因为矩阵乘法的子任务结构中已经显现出一定的依赖特征，也即独立特征，因而具有很高的并行潜力，并且矩阵乘法作为一个反复使用的操作也非常具有优化价值。

子任务具有上述依赖特征的任务，是具有并行潜力的。一个计算任务划分后的依赖关系会限制这个计算任务在具有并行计算能力的硬件上的运行潜力，比如在GPU上或在FPGA的软核上。例如，一个由N个函数运算串行构成的计算任务，其中每个函数运算都需要使用上一个函数运算任务的结果，假设不使用数学手段对整个计算任务进行重构式的优化，那么这N个函数运算任务就完全无法并行处理，只能按照先后依赖次序顺次计算。通常的计算任务的依赖关系并不会如此严苛地形成线状结构，而是会构成一个有向无环图（DAG）。对有向无环图的遍历可划归到流程控制任务领域，这是CPU更擅长的。

### 2.2.2 CPU与GPU

日常生活中最常见的一个异构计算的例子就是CPU与GPU的协同处理。CPU将一些图像计算任务划分给GPU进行运算，GPU在结束其接到的计算任务后将结果返还给CPU，或者直接递交给显示设备，最终将图像显示在显示设备上。这样做的好处主要在于时间效率。图像计算任务中包含了大量的向量、矩阵的浮点计算，具有高并行度的特点。而CPU作为线性指令的处理器，更擅长进行流程调度，而非并行计算。GPU本质上是一种专门用于处理图像计算任务的专用集成电路，本身就具备了较高的并行度，可以在同一时间内执行更多的运算处理任务，因而在执行向量计算任务上，它的效率比CPU要高许多。

CPU（中央处理单元）的主要部件有算术逻辑单元（ALU）、中间寄存器（IR）、运算累加器（ACC）、描述字寄存器（DR）和B寄存器，以及各种总线设备和控制元件。CPU通常采用冯诺依曼架构，有的也采用哈弗架构。冯诺依曼架构主要通过流水线的形式，将CPU处理计算任务时所使用的指令划分为五个子步骤，分别是取指令、指令译码、指令执行、访存取数、写回五个阶段。

GPU（图形处理单元）在结构上与CPU较为相似，其基本构型仍然是线性计算队列的处理。与之不同的是，GPU采用的计算单元是流多处理器（SM），其中每个流多处理器又由多个流处理器（SP）组成，亦包括其它资源，如存储资源等。GPU之所以能够达到比CPU更高的并行性，是因为它下属的流处理器之间具有较高的并行度。面对一个计算任务流，GPU的众多流处理器将计算任务拆分，并分解成为相互独立的计算任务。这些相互独立的计算任务互不依赖，可以在同一时间内以尽可能高的并列状态进行计算，而CPU的处理是没有这一特点的，因而GPU在处理某些计算任务的时候比用CPU处理的效率要高上数倍乃至数百倍。

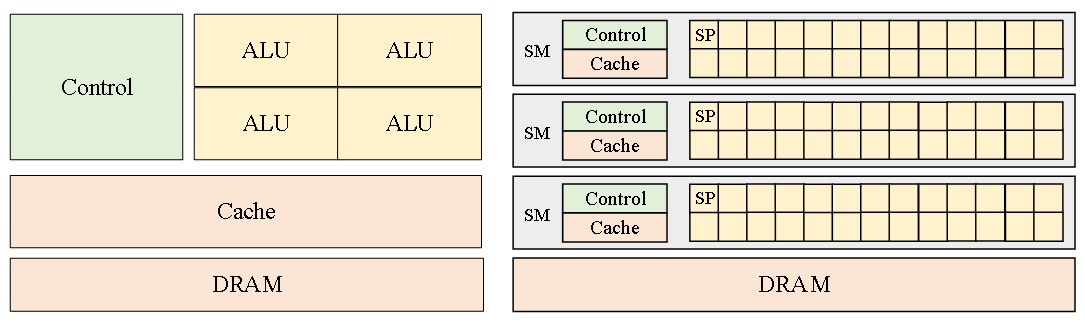
相比GPU的架构，CPU的计算处理单元数目较少，这限制了其并行度的上限。尽管一些CPU架构会使用多核架构来弥补并行度，但通常其核数在数个到数十个。GPU的核数目则远远超过CPU，通常能够到达千量级，而其缓存资源相较CPU更少一些。

图 1 CPU架构（左）与GPU架构（右）示意图

GPU并不是面对所有计算任务都有同样的效能提升。很明显，倘若一个计算任务潜在的并行潜力不足，那么这一计算任务将无法发挥GPU的最大并行效能。当然，倘若GPU的并行潜力不足，那么它也很可能无法发挥计算任务的并行潜力。同样的原理也适用于其它的硬件类型。分配给一个硬件的计算任务要在并行度和结构上进行匹配，才能确保尽可能地不浪费两者的优化潜力。

### 2.2.3 FPGA

起初FPGA作为一种验证性器件，仅在集成电路制造领域的电路验证方面有一些使用热度。之后由于FPGA灵活的可重构性，它的计算性能越来越受到人们重视。直至今日，FPGA在异构计算的领域崭露头角，在一些特殊领域中大有盖过GPU的趋势。

在FPGA之前也有类似的可重构集成电路的概念，即PAL、CPLD等。PAL即Programmable Array Logic，CPLD即Complex Programmable Logic Device，两者的基本思路均为通过具有灵活性的电路构造，等效实现门级网表到实体电路的部署，从而弥补电路设计从门级网表到流片制造中间所消耗的一部分时间投入和成本投入。作为电路验证手段，PAL和CPLD所蕴含的的思想在当时是较为先进的。

FPGA在此基础上更进一步，使用了查找表（LUT，Look-Up Table）来描述电路的输入输出映射关系，并结合触发器对集成电路进行辅助设计和验证。尔后FPGA的一些优秀的计算特性被发掘，开辟了并行计算领域的一大分支。

FPGA在计算上的优秀特性主要在于高并行度、低重构成本和低功耗。虽然由于FPGA走线的限制，为了保证满足功能性线延时需求，其时钟频率不可能像经过反复迭代优化的ASIC那样高，也不及CPU和GPU，但其高并行度的特点足以弥补这一短板，甚至还可能补足短板后还有算力的盈余。CPU和GPU从定义上而言都可以划归到ASIC的范畴，因为它们硬件上不可重构，只能在软件上进行一定的算法调整。那么，一旦生产出来成为产品，CPU和GPU的运算速率的上限就是框定的。例如CPU的运算速率的上确界可以由流水线级数给出，而GPU的运算速率则可以由流水线级数和处理器总数目给出。另外，由于计算任务内部的依赖性，GPU的运算潜力可能被进一步限制。在神经网络部署的情形下，通常GPU的并行度是不足的。神经网络模型巨大的参数量较之通常的图像处理而言要高出几个数量级，较大的神经网络中包含了巨量的张量运算，而张量运算是典型的并行计算任务。事实上，这样庞大的并行需求不仅使得FPGA异构计算成为热点，也催生了GPGPU（General Purpose GPU，通用图形处理器）。

Wim vanderbauwhede曾提出千核处理器，在FPGA芯片上实现了1000个微处理器，从而使得整体运行速率较之传统CPU芯片提高了约20倍。事实证明，FPGA确实可以在计算领域超过传统器件[6]。

但FPGA独立地作为一个可以承担高并行度计算任务的器件还是不够的。不同种类的计算设备各有其强项，而各种集成电路也存在研发周期和市场周期，异构计算技术仍然有潜力发掘出不同架构的硬件各自的能力。因而人们开始研究，如何将CPU和FPGA放在一起运行，将FPGA看作是CPU的单个协处理器或协处理器群，进而使之接收CPU提供的协处理指令，以高并行度的高效方式进行运算处理，最后将结果返回至CPU。

# 第三章 LeNet在CPU/GPU/FPGA上的部署

## 3.1 平台、语言及工具链

### 3.1.1 开发语言

本设计涉及的开发语言有python、C++、基于C++的HLS语法、HTML、CSS、JavaScript、PHP、Windows BAT脚本语言、bash脚本语言。

本设计中python语言用到的较为重要的模块是PyTorch、numpy、matplotlib，其中PyTorch用于搭建深度学习模型，numpy主要用于PyTorch的参数导出，matplotlib主要用于多维矩阵到图像的转换。

本设计中C++语言（不含HLS语法）用于host程序的开发，其中host程序在Vitis框架下开发。

本设计中基于C++的HLS语法用于kernel程序的开发，其中kernel程序也在Vitis平台中开发。HLS是相对于verilog这种底层硬件描述语言而言的对应概念。高层次综合意在将使用C/C++等高级语言开发的程序进行综合，并生成verilog或VHDL等语言的硬件描述文件，以减轻开发人员的设计工作量。但由于硬件开发思路和线性语言开发的思路在并行性上的不同，HLS相比高级语言常常需要使用一些标记来明确高层次综合工具的综合方向，例如循环展开或流水线展开。

本设计中HTML语言、CSS语言、JavaScript主要用于前端的开发。其中HTML用于网页组件的静态布局，CSS作为一种静态样式标记语言，用于定义网页的静态风格，JavaScript用于定义HTML组件的响应动作。其中在JavaScript中主要使用到jQuery模块，可以更方便地进行ajax报文的发送。

本设计中PHP语言主要用于后端的开发，主要功能是接受JavaScript脚本发送的ajax报文并予以回复。

本设计中Windows BAT脚本语言主要用于在windows操作系统中切换conda虚拟环境并启动LeNet部署程序。

本设计中bash脚本语言主要用于在linux操作系统中切换conda虚拟环境并启动LeNet部署程序，以及进行Vitis平台下的模型自动验证和部署。

另外，本设计使用了conda工具来管理python包的安装环境。Conda通过创建不同的python虚拟环境来兼容不同版本的python包，因为它们可能对python的版本有不同的依赖关系。对于可移植性和兼容性而言，这将是很重要的工具。

### 3.1.2 PyTorch深度学习框架

PyTorch是python语言下的热门深度学习框架。相比同类型的TensorFlow框架，pytorch的编程接口更简洁方便。

一个较为简单的例子是，基于pytorch的nn.Module基类新建一个自己的网络类，并重载nn.Module的forward方法。Forward方法接受一个tensor类型的变量作为输入，并且输出一个tensor变量。一旦定义了这个forward方法，在灌入输入数据后，网络会自行记录对应位置上的梯度，以便优化器根据自定义的损失函数进行优化迭代。

### 3.1.3 Vitis硬件部署平台

Vitis是Xilinx公司开发的硬件部署平台，其中包含的组件有Vivado、VivadoHLS、Vitis AI、Xilinx Runtime Library等。本设计主要使用的硬件平台即Xilinx公司设计制造的Alveo u280数据库加速卡，使用Vitis平台可以较好地部署实例。

使用Vitis部署实例的方法可以分为命令行部署方法和用户图形界面部署方法两种，本设计主要用到的是命令行部署方法。其流程主要是，编写基于C++语言的host代码，编写基于C++的HLS代码，并依次进行软件仿真、硬件仿真和硬件部署。在软件仿真、硬件仿真的步骤，host程序除了承担内存分配、内存转移、流程控制等任务之外，还需要进行数据验证，以确保host程序和kernel设计的正确性。

以上过程均可通过bash脚本完成。编写一个bash脚本文件用于自动化设计验证部署是有必要并且并不困难的。

## 3.1 基于DNN的MNIST识别程序

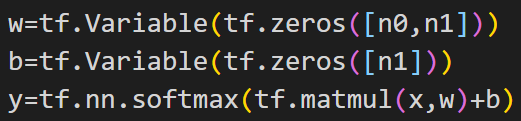
本设计计划从较简单的网络入手，实现面向深度学习的异构加速计算部署。初期计划通过python-tensorflow实现一个MNIST手写识别的DNN网络，并部署在FPGA板卡上，测试其效果，后续再尝试部署一个简单CNN网络。

在后续的工作中发现了该DNN网络的种种问题，以及CNN的种种优势，故而最终的方案中全面选用了CNN网络（LeNet网络）。但即便如此，前期在DNN网络上准备工作也确实有许多值得记录的地方。

### 3.1.1 DNN网络设计与训练

现阶段MNIST手写识别选用一个单隐含层的全连接网络，激活函数选用softmax函数。Softmax函数的表达式为，严格来说施加在网络中的softmax函数并不是激活函数，而是一个带有归一化性质的函数。

涉及softmax的核心代码如图：

图 2 DNN网络设计中涉及softmax的核心代码

其中n0是层输入规模，而n1是层输出规模。权重w和偏置b取全零为初始值。y为该层输出，用数学语言描述其中关系即为，，其中z在代码的嵌套中被隐去。由于yi的值不仅需要参考zi的值，还需要参考其它zj的值，因而说此处使用的softmax并不是通常意义上的激活函数。

由于MNIST数据集中单个手写图像的尺寸为28x28=784，输入层的规模取为784。由于MNIST手写识别的目标是判断一张图片是数字几，因而输出层的规模取10，使用one\_hot编码。在此基础之上，取隐含层规模为100。

损失函数取交叉熵，即，其中ylabel是测试集中给出的label，而y是网络根据输入得出的判断结果。

优化算法选取的是梯度下降算法，即GradientDescentOptimizer。

### 3.1.2 网络的修改

上一节展示的是最初预想的设计，真正训练过程中曾经遇到两个棘手的问题，最终均得到了解决。

第一个问题是启动失败问题。在最开始，权重w取的是全零初始值，这样会让所有层的输出值在最开始为零。而神经网络的参数优化选取的是梯度下降算法，因而网络优化无法启动，无论轮环训练多少次，损失函数始终维持在较高的水平。后修改关键部位代码，将“w=tf.Variable(tf.zeros([n0,n1]))”修改为“w=tf.Variable(tf.random\_normal([n0,n1],stddev=0.1))”，其中权重w取标准差为0.1的正态分布的随机数为初始值。修改后网络优化得以启动。

第二个问题是准确率上限问题。训练过程中设定每轮进行epoch=2000次训练，每次训练取100组数据。训练停止的判断标准为连续4轮训练中，识别的精度波动保持在epsilon=0.001以下。在多次训练过程中，发现训练结果差异巨大，随机性极强，并且准确率始终保持在较低的水平。

表1 对出现精度问题的DNN网络进行测试

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 次数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 精度 | 54.5% | 23.6% | 11.9% | 38.0% | 57.7% | 33.5% | 46.2% | 61.3% | 29.1% | 50.9% |

其中准确率最高的一次是61.3%，最低的低达11.9%。这说明神经网络设计上出现重大失误，查询各大网站博客均未能寻到有效的解决方法。后经过代码检查得知原因是数据集未乱序给入。由于tensorflow的较新版本中移除了旧版本中内建的input\_data函数，并建议用户自行设计数据给入方法，因而我自行按照理解编写了input\_data函数，从MNIST数据集中取出数据，并依次轮环给入神经网络用于训练。该原因是偶然起意，试图在旧版本tensorflow（1.13）环境下运行相同代码（除自定义的input\_data方法外），结果发现精度迅速升值75%以上，并在更长时间的训练后到达89%才发现的。查看tensorflow源码确认后得知，tensorflow原生代码中的input\_data是使用了shuffle方法对数据集进行打乱的。此前从未意识到数据集顺序会对神经网络的最终效果造成影响。

在自定义input\_data方法中加入shuffle后，再次进行了十次测试，以三次训练停止为目标，即当网络优化过程因连续4轮训练识别精度波动保持在0.001以下而自行停止后，再次启动，直至自行停止满三次。精度记录如下：

表2 对修改后的DNN网络进行测试

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 次数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 精度 | 86.7% | 90.5% | 74.4% | 92.0% | 87.2% | 89.8% | 88.5% | 88.1% | 85.3% | 89.5% |

虽然这一DNN网络并不是最终实例中使用的，但这些经验对后续的工作是有必要的。实际上，后来所用的LeNet中也包含两个全连接层，而这正好是弃用的DNN网络的缩影。

## 3.2 使用PyTorch在CPU/GPU上部署LeNet

### 3.2.1 LeNet网络结构

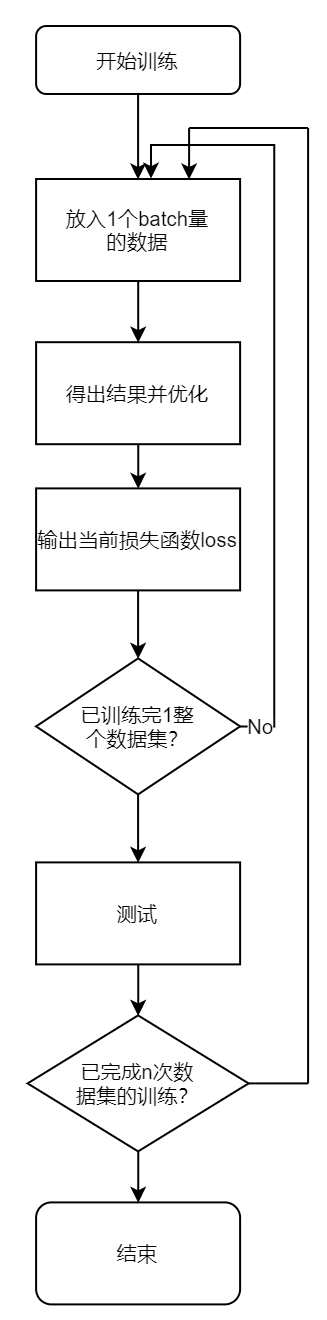


图3 LeNet结构图

LeNet主要由两个“大卷积层”和两个全连接层组成。此处将一个卷积层和一个用于降采样的池化层合称为一个“大卷积层”。之后的高斯连接层是一个线性连接层。

第一个卷积层对尺寸为32×32的图片进行6次卷积操作，得到6个通道的数据。在应用于MNIST数据集时，由于MNSIT数据集中的图片数据尺寸均为28×28，因而对第一个卷积层进行稍许修改，为其加上尺寸为2的padding，便可以将28×28大小的图片等效外扩到32×32的范围。该卷积层后的池化层采用步长为2的池化操作，将每个通道的图片压缩成14×14的大小。

第二个卷积层对尺寸为14×14大小的图像信息进行16次卷积操作。由于第二层的输入本身就是6通道的，因而每个卷积核都是三维的。经过降采样操作，数据变为16通道的5×5信号。

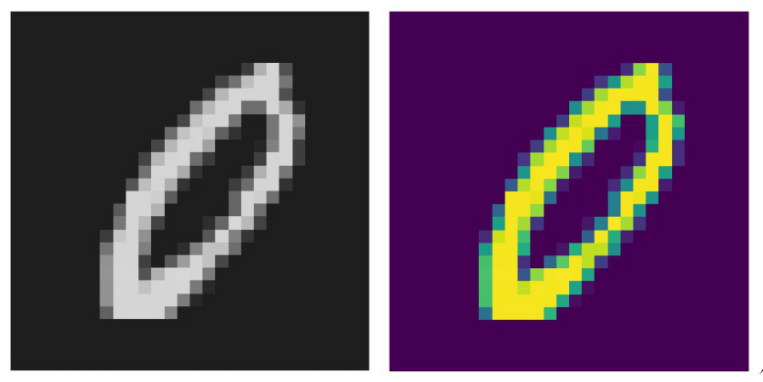
此后连接两个全连接层，其中全连接层的激活函数选用ReLU函数，这对硬件实现较为友好，因为相比此前使用的softmax和sigmoid函数，ReLU函数更容易硬件实现。例如要在硬件上实现sigmoid函数，最佳的选择可能是cordic类似的算法，这会对硬件设计带来较大难度。相比之下，用一个正负判断完成的ReLU操作，无疑减轻了实现难度。再加上ReLU在防止训练后期优化停滞问题也颇有成效，故选用ReLU函数作为激活函数应为明智之举。最后的高斯连接层将识别结果映射到0~9的判断结果上。

图4 LeNet程序核心流程图

### 3.2.2 代码编写

未添加任何其他功能的LeNet训练程序流程图如下。只需事先给定需要训练的轮数和每一个批次的数据量，依次给入神经网络的入口，并调用事先设置好的优化器，即可让网络参数自行优化。之后的附加功能均基于此实现。

附加功能主要有，将损失函数loss和其他信息输出到log文件中，将当前正在训练的图像进行存储，保存当前网络参数，读取已保存的网络参数等。

存储当前正在训练的图像主要用到matplotlib模块中的pyplot模块，该代码包多用于数据处理，也可用于显示图像。其特点在于能够较为方便地处理F模式的图像，即二维无通道图像。若使用Pillow模块进行二维无通道图像的存储，则需要先将图像转换成L模式、RGB模式或RGBA模式，并且最终得出的图像为灰度图像。而使用pyplot则默认会输出伪彩色图。

保存和读取当前网络参数主要使用pytorch内置的save方法和load\_state\_dict方法。这两种方法适用于仅保存网络参数而不保存网络模型的情况。

图5 左：原图 右：matplotlib保存的伪彩图

## 3.3 使用VITIS在FPGA上部署LeNet

将LeNet部署在FPGA上的过程和在CPU/GPU上的迥异。本设计计划使用Vivado HLS，在FPGA平台上部署一个LeNet，并设计其和网页的交互方式。

### 3.3.1 LeNet网络参数的获取

使用FPGA进行网络参数的优化训练不是明智之举，因为FPGA的存储模式限制了优化器寻找前级的梯度，要搭建专门适用于网络参数优化的电路设计又十分复杂，并且在电路上有较高的冗余度，总体而言在时间上和在电路面积上并不经济。因而本设计在FPGA的网络部署中，直接取用CPU/GPU版本训练得出的优化结果。

这一思路遇到的一个问题是，CPU/GPU版本的使用的是pytorch，而Vivado HLS工具支持的是C/C++语法和语言标准库，因而Vivado HLS工具无法直接取用pytorch存储的数据，中间的数据需要转换，不能使用pytorch内置的方法。针对这一问题，本设计中采用了简洁的文本形式存储。

Pytorch内置了numpy方法，可将Tensor类型的数据转换成ndarray格式的数据。在此会遇到两种特殊情况，一是某个Tensor是存储在GPU中的时候，需要用torch.Tensor.cpu()方法将该Tensor导出到CPU的存储空间中，另一个特殊情况是某个Tensor需要grad（梯度）的输入参数时，numpy()方法无法被正确调用，此时需要先使用detach()方法将参数从网络中脱落，相当于一次仅数据的拷贝，从而正确地转换为ndarray格式。

为了方便FPGA上的网络部署，从CPU/GPU版本导出的文本形式的网络参数将在文件中以每行一个数据的方式存储，类似于内存空间。并且，网页系统上将添加一个功能，即每次在CPU/GPU实例中保存网络参数时，不仅保存为pytorch格式的存档，还保存一份用于FPGA网络部署的文本格式存档。

### 3.3.2 网络结构的设计

用于部署在FPGA上的LeNet网络结构（以下简称LeNet-FPGA）不像使用Pytorch部署在CPU或GPU上的LeNet网络结构那样具有方便灵活的编程接口。本设计采用的方式是，网络结构并不包含参数权重，而仅具有运算功能，参数权重以文本的形式存放在文件中，由host程序将文本中的数据转移到内存空间，进而协同运算。

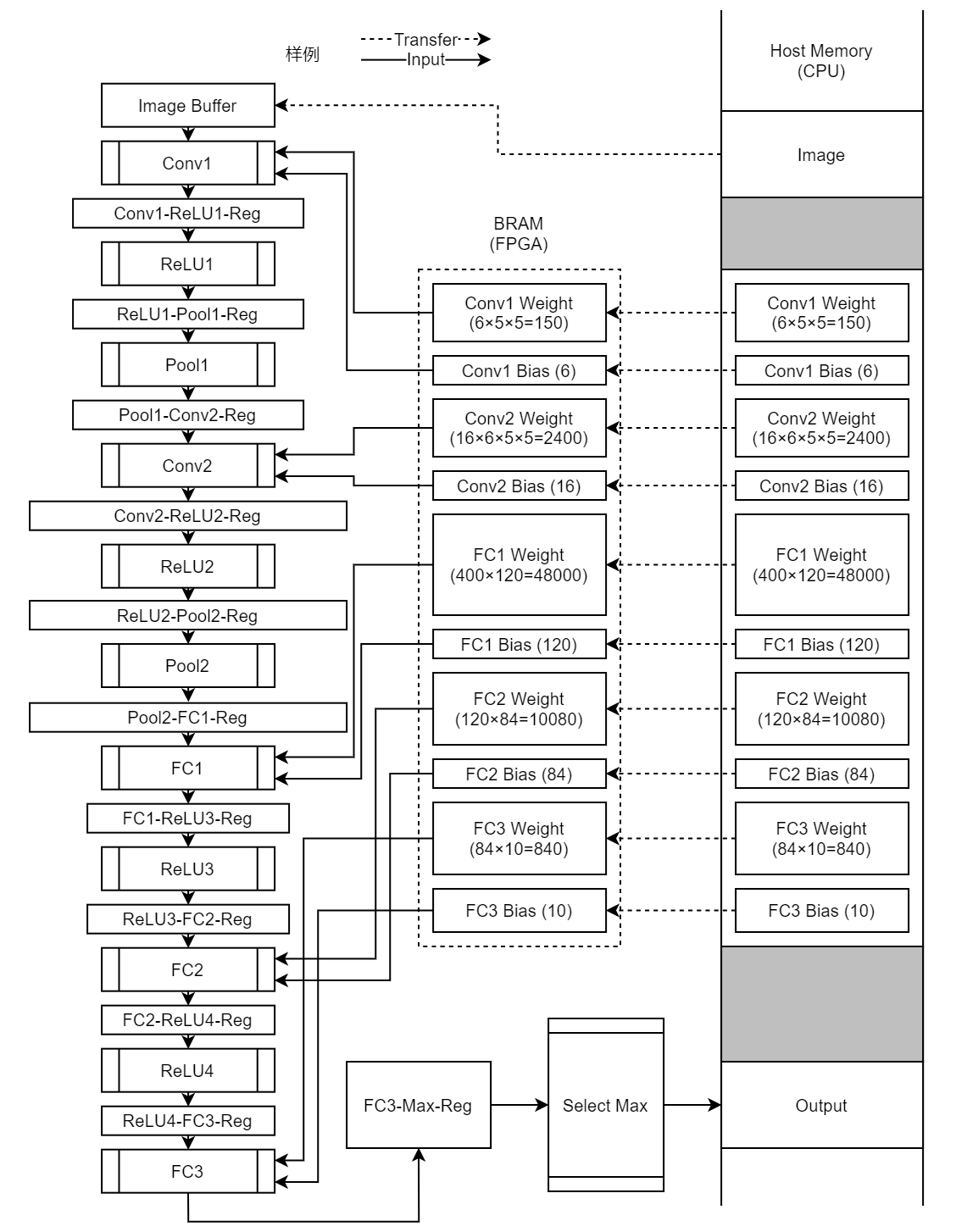
HLS模型要求内存空间尽可能给定深度，这对于已经确定了的网络模型和网络参数并非难事。将图片考虑成32×32的大小，则占据1024个float类型的内存空间。LeNet-FPGA的第一个卷积层的卷积核有6个，每个卷积核大小为5×5，因而占位为150，伴有大小为6的偏置。第二个卷积层有16个卷积核，每个卷积核大小为6×5×5，因此占位为240，伴有大小为16的偏置。第一个全连接层的输入大小为16×5×5=400，输出大小为120，因而占位48000，伴有大小为120的偏置。第二个全连接层的输入大小为120，输出大小为84，因而占位10080，伴有大小为84的偏置。第三个全连接层（线性层）输入大小为84，输出大小为10，因而占位840，伴有大小为10的偏置。以上参数量加上图片的存储空间，总计61706。

图6 LeNet 核 结构关系图

以上参数均通过定义好的axi总线传递到HLS模型定义的硬件电路，LeNet-FPGA电路本身只需对参数进行运算即可。LeNet-FPGA的设计中在每两个神经网络层之间安置了数组以存储运算信息，这在最终综合成rtl代码时会成为寄存器。

左图为host与kernel之间的工作模式。Host事先准备好三块内存，一块是表示图像的Image区，大小为32×32；一块是表示参数的Param区，连续分布了LeNet需要用到的所有参数；一块是用于存放输出结果的Output区。

Kernel预先把参数拷贝到FPGA内部的存储空间中，以便计算时更好地发挥FPGA的并行优势。在准备好参数后，数据流便从host memory中的图像区按图中所示顺次运算，直到获得最终结果。

此设计的并行性潜力主要在于，其中卷积层、池化层、ReLU激活层和线性全连接层中有较多互不依赖的计算任务，相比CPU/GPU版本的训练可以展开循环。

### 3.3.3 网络结构的验证

由于Vitis 和 Vivado HLS支持的是c/c++语言，而LeNet网络参数是从pytorch中得来的，中间存在经过文件的转储过程，这样得出的数据就可能存在顺序问题，即传出的参数可能倒置或错位，这将会对最终的结果识别产生严重影响。遵循少量多次的设计修改原则，设计过程采用多个轮次，第一个轮次实现第一个卷积层，并对照CPU版本的LeNet程序验证该层输出，之后每个轮次在原有基础上再添加一层，并验证层输出是否正确。

实际操作过程比想象中更顺利，这是因为PyTorch采用的张量策略与numpy模块中多维矩阵的策略一致，其数据线性存储部分和形状特征是分离的，因而数据之间的顺序不会随意更改。

# 第四章 面向LeNet实例部署的前端设计

## 4.1 总框架结构及其实现

为了模块化地设计各个页面，本设计中使用了较多的iframe元素。所谓iframe即html内联框架，可以在一个页面中引用另一个网页作为其中的一个子块。

本设计中的布局主要是，主页面包含三个标签，以及标签对应的页面。这三个标签分别是主页、任务视图、shell。主页上主要放置文字说明，任务视图用于LeNet实例的部署和各种命令的收发、运行状态的查看等，而shell则作为一个保留功能，用于直接在网页端发送shell命令，意图用于更个性化的部署命令，之后可能会考虑到安全性问题做出框限或封锁。

图7 前端框架关系图



以CPU版本的LeNet部署监测页面为例，其展示的最终效果如图所示。图中展示的是网络部署程序开启但不在训练状态时的展示效果。其中stop按钮用于停止网络部署程序，load用于加载网络参数，save用于保存网络参数，train用于让部署程序进入训练状态。下方的文件拖拽识别区域接受一般的图片类型，拖拽到区域后LeNet部署程序会进行一次识别，并将识别结果放在log文件中。Log文件的最新8行会显示在四个按钮的下方，例如图中显示出的idle状态信息。

图8 拖拽识别功能示意图

## 4.2 监测页面的设计与实现

在使用PyTorch作为支持的情况下，CPU版本的LeNet的监测页面、运行代码和GPU的非常类似，FPGA版本的LeNet部署中使用的host程序是在CPU架构下完成的，有差异的只是LeNet的核心运算，因而CPU、GPU、FPGA三者在网络响应的设计上基本是一致的。

### 4.2.1 前后端响应

需要设计的响应事件有：开始部署，结束部署，加载网络参数，存储网络参数，开始训练，查询当前正在训练的图像，查询当前网络部署状态。

由于网页前后端程序不可能和LeNet真正的部署程序耦合得很紧密，网页前后端能够实现的功能大部分是查看训练状态，以及放置代表指令的信标文件，因而在前后端响应的部分只需要着眼这一点即可。

“开始部署”的实现与其他几个功能的实现有所不同。以CPU版本的前后端响应为例，当html页面上的按钮按下，浏览器开始执行CPU\_start()函数，其中包含了向LeNet\_start.php发送的一个ajax报文，报文中标明了CPU标签，意味着这是用于CPU版本的LeNet启动信号。当后端的LeNet\_start.php接收到该报文时，会解析该报文并识别出这是用于CPU版本的LeNet启动信号，进而调用LeNet\_CPU\_start.bat脚本来启动用于CPU版本LeNet训练的python脚本。

最初设计实现的时候，是没有LeNet\_CPU\_start.bat脚本的，php脚本接收到报文并解析后会直接调用python脚本，但这样一来会面临另一个问题：由conda配置的虚拟环境无法生效。为了兼容多个版本的tensorflow、pytorch、cuda等python支持包，安装conda（anaconda或miniconda）或其他虚拟python环境搭建工具时必要的。但即便已经安装conda，php脚本调用python时会默认进入base虚拟环境，无法指定新建的虚拟环境。本设计中的网页最终目标是作为异构部署平台的一部分，而无法指定虚拟环境对于该平台的通用性和移植性是有害的。为解决这一问题，使用bat脚本的时候，可以通过CALL <conda path>\Scripts\activate.bat <conda virtual env path>来修改脚本的运行环境。

Bat脚本是windows操作系统的常用批处理脚本。同样在windows下使用的还有PowerShell脚本，但PowerShell脚本相比bat脚本而言还不够成熟，向下兼容性也不够好，因而本次设计暂时未设计对PowerShell的支持。在linux系统中，使用sh脚本同样可以达成虚拟环境转换的功能。

“结束部署”、“加载网络参数”、“存储网络参数”、“开始训练”这四个指令，加上对当前正在训练的图像的查询、对当前网络部署状态的查询功能都使用了信标文件的概念，涉及进程间通信，放在下一小节中详述。

另有一个功能，是获取网络部署过程中的log信息，并选取最新的8行输出到网页上，该功能较易实现，不再赘述。

### 4.2.2 进程间通信

这里的进程间通信主要指php用于响应前端ajax请求的进程和用于训练LeNet的python进程之间的通信。这一过程的主要设计思路是信标文件。

信标文件的设计思想取自分布式计算中的“全局存储空间”的想法。由python写就的程序进程和由php写就的程序进程间直接通信较难实现，这需要更底层的实现，超出了我目前的能力范围。但同时可以观察到，python进程和php进程间的通信壁垒犹如分布式计算中两个处理器之间由于端口设置而造成的直接通信壁垒一样，这时候借鉴分布式计算中会采用的“公用空间”的方式，可以在文件系统中创建一个cache文件夹，用于存放LeNet程序进程和php后端响应的共用内容，例如当前的运行状态，或者需要LeNet程序执行的指令，从而实现两者间接的通信交互。

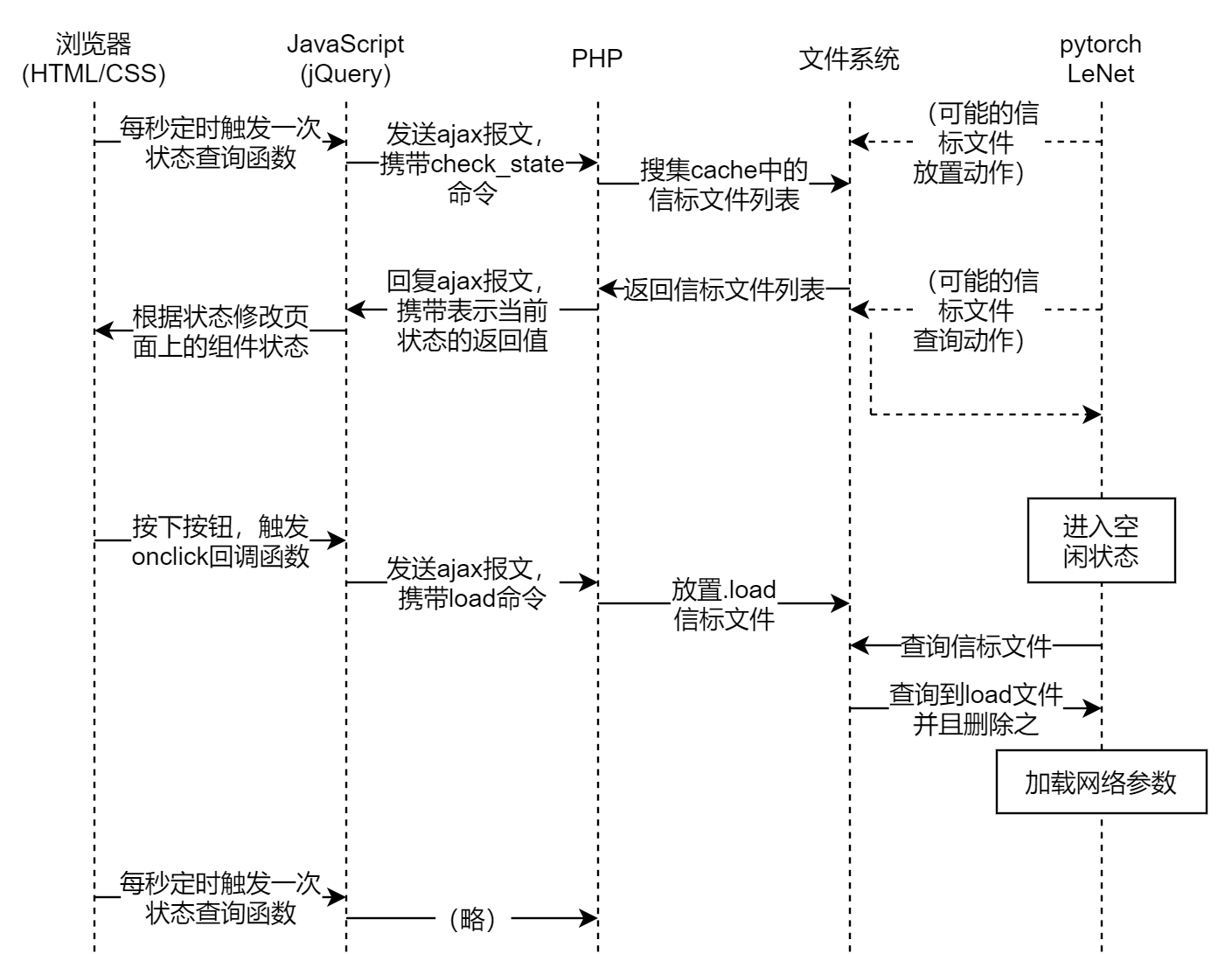
另有一个关键性的设计，那就是独占原则的实现。本设计中CPU/GPU版本的LeNet程序一经bat脚本的调用打开，便在后台无输出地工作。倘若同一时间有不限数量的LeNet进程运行，对于服务器的负荷是巨大的，并且这些进程的关停也会存在问题，总而言之那样的设计是有重大缺陷的。因而必须要设计一个机制，使得同一时间内机器上只有不多于一个CPU\_LeNet进程，不多于一个GPU\_LeNet进程。

图9 前后端响应关系图

借助信标文件，这是不难做到的。以CPU版本为例，只要当相应的python程序开始时，在cache文件夹中放置一个LeNet\_CPU.running信标文件，表示当前正有一个CPU版本的LeNet实例处于运行状态，那么再有第二次“开始部署”的ajax请求发送到php脚本时，php脚本所调用的bat文件就会先检查是否已经有一个同类型实例处在运行状态，若有，直接返回，不再开启新的实例。当然，当一个实例正常退出时，

类似地，当php脚本接收到“结束部署”、“加载网络参数”、“存储网络参数”、“开始训练”命令时，也会创建对应的信标文件。

不创建信标文件的命令只有一个，就是“检查实例运行状态”。该命令会让php脚本收集当前是否存在.running、.training的信标文件，并以返回值表示当前状态是未部署、空闲还是训练中。

若平台遇到某些未知因素遭遇严重的崩溃，以至于不知道有没有实例运行在后台时，可以手动创建同于停止的信标文件，用于停止实例。

值得注意的是，“开始部署”和“开始训练”是两个不同的指令，这与LeNet部署在CPU/GPU上的代码中的控制流框架有关，在下一小节中会展开解释。

前后端响应和进程间通信的过程如图所示。该示意图以每秒定时触发的状态查询动作和load按钮为例，介绍整个系统的工作方式。

每秒定时触发的状态查询动作中包含了一个携带check\_state命令的ajax报文，该报文发送到相应php脚本后，php脚本会搜集cache路径中的信标文件列表，并将搜集的结果处理后返回给前端的页面，例如查询到了.running信标，但没有查询到.training信标，则说明程序已开启，但处于空闲状态，返回值为1。然后前端的回调函数根据该返回值，对页面上的组件进行显示或消隐，以免错误地触发按钮的onclick函数。

而以load按钮为例，按下load按钮将会触发按钮绑定的onclick回调函数，该函数发送一个携带了load命令的ajax报文，该报文传递到php脚本后会令其放置一个.load信标文件。该回调函数仅发送报文，不像每秒定时触发的动作中需要php脚本的回复。该.load信标文件并不立即生效，而是等待进入空闲状态的pytorch LeNet程序主动查询cache中的信标文件。Pytorch LeNet程序的流程将在下一小节中详细阐述。

### 4.2.3 各版本LeNet的代码调整

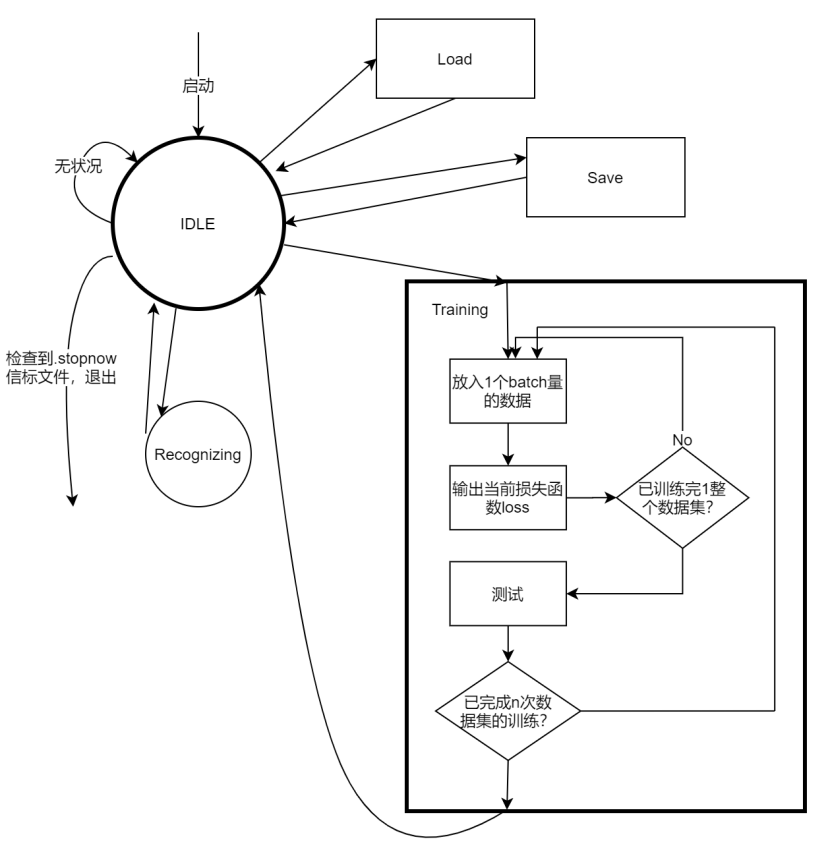
之前实现的主要是最核心的LeNet功能，但这些功能还需要根据上节所说的集散式信标命令系统进行调整。最终成型的程序流程图如下。程序启动后直接进入idle（空闲）状态，并在此状态下反复查询cache路径中有无代表命令的信标文件，时间间隔为1秒。若查询到.stopnow文件则立即停止运行，若查询到.load文件则加载默认路径下的网络参数，若查询到.save文件则将网络参数保存到默认路径下，若查询到.launch\_training文件则开始训练，并放置.training信标文件，表示当前程序正在训练中。训练状态下程序不查询信标文件，因而只能等待程序训练自行结束，回到IDLE状态方可继续响应。之后若有添加强制结束训练的功能，亦可以较为方便地实现。

图10 LeNet程序状态跳转图

FPGA版本的代码与此图所示的流程有所差异，主要保留了IDLE、Load和Recognizing三个状态，并且Load功能可以随时取用CPU版本或GPU版本LeNet保存好的训练参数。

### 4.2.4 页面崩溃问题的解决

在LeNet的查看页面的设计实现过程中，遇到了两次原因不同的崩溃问题，最终均得到了有效解决。

第一个崩溃原因是php响应时间限制。由于LeNet的部署训练通常需要较长的时间，使用php脚本打开bat脚本再打开python程序时，由于python程序迟迟不退出，php脚本便会一直计算响应时间，当python程序退出时php脚本便会认为执行超时，并引发响应错误，致使页面崩溃。为解决这一问题，可在php脚本调用bat脚本之前修改其计时参数，使其不限时间。这一修改不会影响其他指令的响应时间。

第二个崩溃原因是同步响应的问题。Ajax报文同步响应的功能在XHTML的新版本中不被推荐使用，并且报文强制同步发送、响应确实会造成一些线程调度上的问题。在实现过程中表现为，虽然开始部署、读取参数、保存参数、开始训练等功能均可以较好地实现，但停止部署的功能实施后，经过3~5分钟的等待，页面通常会崩溃。为解决这一问题，只需在页面加载时加上一句“$.ajaxSetup({async:true});”设置命令，令全局的报文都加上异步收发的设置。当报文的发送规则从同步变为异步时，其后面的命令也由阻塞变为非阻塞，即其后的命令无需等到报文响应才执行。该命令需要jQuery支持，并且仅对jQuery的ajax报文命令生效。

### 4.2.5 图像刷新问题的解决

在实现过程中遇到的另一个问题是图像的刷新问题。CPU和GPU版本的部署监测程序包含了查看正在训练中的图像，具体做法是不断更新名称相同的图片文件。但浏览器的策略之一是，一旦一个img元素从服务器端请求了一张图片，之后除非该img元素所绑定的src（源）URL发生变化，都不会再向后端请求该链接对应的图片。因此，每次使用JS语句试图通过给相应img元素的src属性重新赋值的方式并不会造成图像的刷新，因为前后两次src属性并没有发生变化。

解决方案是，每次需要刷新img元素时，对其src属性赋值时加上时间戳，例如:

“$('#image').attr('src',"training.jpg"+"?random="+new Date().getTime());”

在添加了时间戳后，浏览器将会认为这是不同的链接，并且会向服务器再发送一份请求来获取图片，进而成功刷新页面上的图像。

# 第五章 测试与前端效果

## 5.1 前端效果一览

图11 前端效果：部署前



图12 三版本的LeNet均开始部署，并处在idle状态



图13 加载已训练好的模型参数

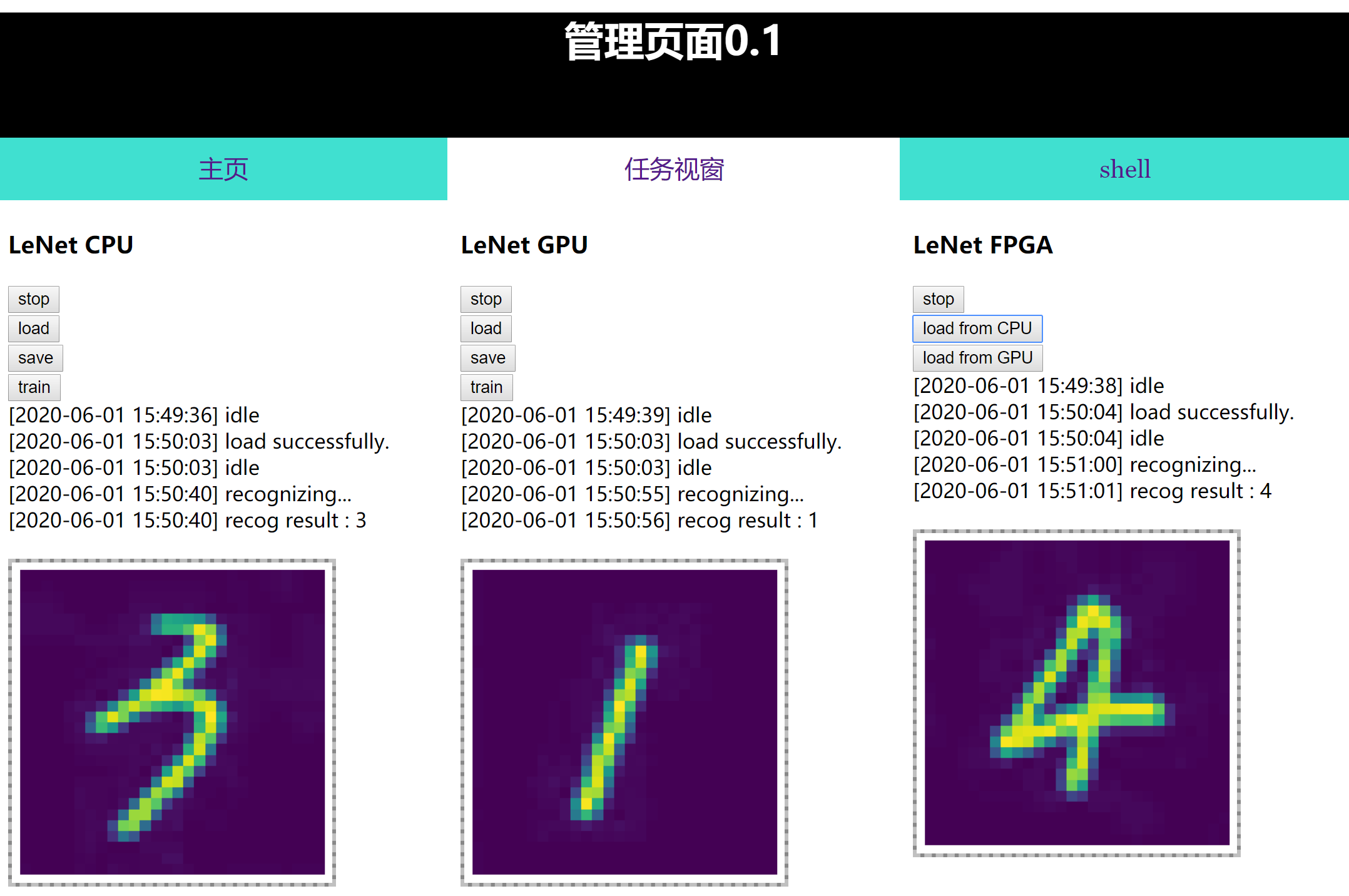


图14 拖拽图片文件并进行识别

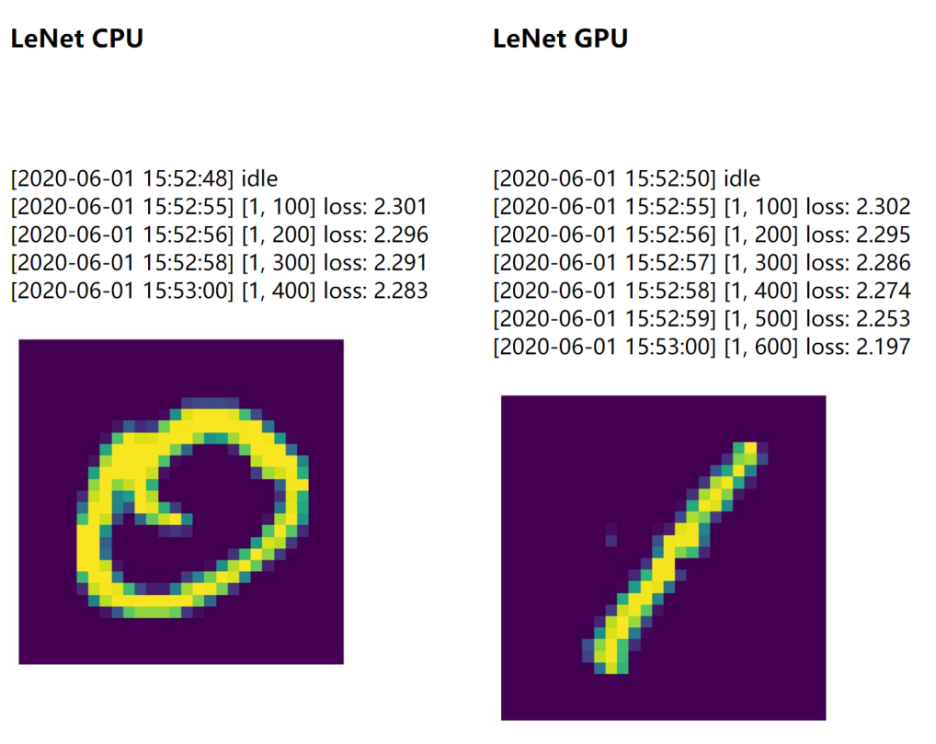


图15 从头开始训练

如图15所示，Log中显示的训练过程中的损失函数值，下方显示的是训练中的图像。

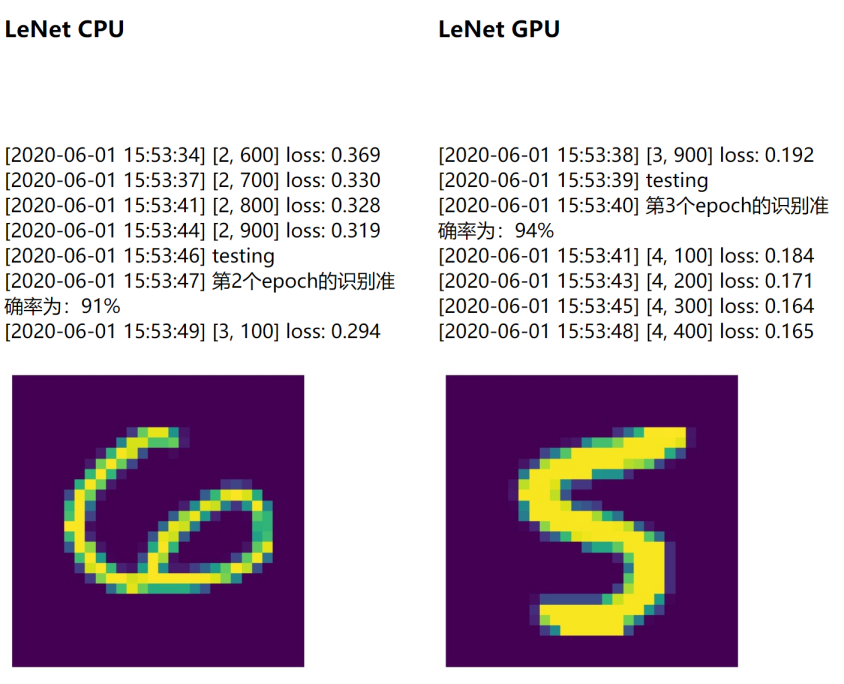


图16 每epoch测试的识别结果

如图16所示，由于CPU版本和GPU版本的程序启动时间基本相同，从log信息中不难看出，GPU版本程序的训练要比CPU版本的快许多。最终查看log文件可知，CPU版本训练16个轮次共需要568秒，GPU版本训练16个轮次共需要384秒，两者训练速度比例为1：1.479。

## 5.2 CPU/GPU/FPGA版本LeNet的识别率

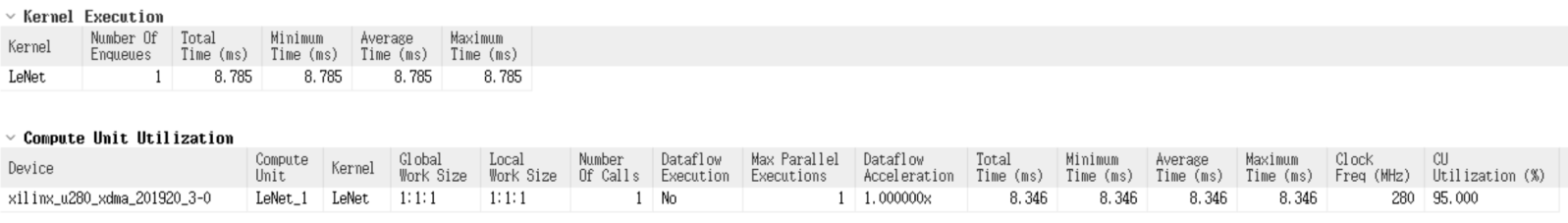
在16轮训练后，CPU版本的LeNet程序识别精确度可达97.8%，GPU版本的LeNet程序识别精确度可达98.0%。直接使用CPU、GPU版本训练数据的FPGA版本LeNet核的识别精确度均达到96.7%。CPU/GPU的测试精度差异非常小，属于正常现象。其中FPGA精度略低于CPU/GPU的成因应为量化误差。

表3 前人类似工作及其精度

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 文章 | 网络类型 | 数据集 | 特殊方法 | 识别率 |
| 王风盼 [基于深度学习的手写数字识别方法研究](http://kns.cnki.net/KCMS/detail/detail.aspx?filename=1018854662.nh&dbname=CMFD201901&dbcode=cdmd&uid=WEEvREcwSlJHSldRa1FhdkJtNEYwV2QwK1FBK09wWHlBdFIvb2QwN1A4WT0=$9A4hF_YAuvQ5obgVAqNKPCYcEjKensW4ggI8Fm4gTkoUKaID8j8gFw!!&v=MzI3NTF0RmlEaFViM01WRjI2RnJ1OUd0ZktyWkViUElSK2ZuczR5UllhbXoxMVBIYmtxV0EwRnJDVVI3cWZZZVo=" \o "基于深度学习的手写数字识别方法研究" \t "http://kreader.cnki.net/Kreader/_blank)[7] | LeNet | MNIST | 高斯噪声 | 99.23% |
| 刘辰雨 基于卷积神经网络的手写数字识别研究与设计[9] | LeNet-5 | MNIST | 无 | 93.61% |
| 李倩倩 基于深度学习的手写体数字序列的识别研究[10] | LeNet-5 | MNIST | 无 | 98.96% |
| Wu L, Zhu Z. Towards understanding generalization of deep learning: Perspective of loss landscapes[11] | LeNet | MNIST | SGD | 97.90% |

参考其他几篇文章中的识别率，本设计达到的精度处于中游水平。

CPU版本的单次识别响应时间为2.992ms，GPU版本的单次识别响应时间为1.995ms。参照图17提供的数据，FPGA版本的单次识别响应时间为8.346ms，相对来说劣于CPU和GPU版本，但处在同一数量级。其中原因有三，一是因为FPGA本身主频的劣势，这会导致单位时间内FPGA能够执行的拍数少于CPU/GPU能够执行的指令数；二是本设计中LeNet的FPGA核循环展开不足，结构上仍有较大的提升空间；三是LeNet核在搬移数据时可能花费了过多时间。



另外，在鲍云峰的一篇文章[8]中，其基于OpenCL设计的图像识别模块的识别延时可低达0.455ms，这意味着本设计中的计算核在延时表现上尚有很大提升空间。

图17 FPGA版本LeNet运行记录

# 第六章 工作总结和展望

## 6.1 工作总结

本设计主要完成了一个面向LeNet深度学习模型的前端网页设计，并在CPU、GPU、FPGA上分别部署了相同的LeNet模型，并比较了它们的精确度。另外，测试了FPGA上部署的核的响应时间。借由本设计，对深度学习有了更深的理解，对异构计算也有了一定的实践经验，并对前端设计有所涉猎。

## 6.2 后续工作展望

本设计还可以在以下方面进行功能的增添和修缮。

灵活性方面。本设计暂时只能人工进行单种网络（LeNet）的部署，这是不够灵活的。后续可以添加一个功能，在网页上确定神经网络的架构，选择优化器等策略，并一键生成部署文件，灵活地部署到三种架构的硬件上。面向多样的神经网络设计部署工具将会用到Vitis AI库，这是Xilinx公司的又一力作。

一键部署的自动化设计虽然让用户获得了便捷的体验，但同时也限制了用户的设计自由度。基于这一考虑，本设计原本准备了可直接运行shell命令的页面，但这一部分的设计还很不完善。一方面，直接接受网页上运行的shell命令存在较大的安全问题，而为了易用性牺牲整个平台的安全性是得不偿失的。另一方面，这一页面对shell的支持也不完善，例如cd语句并不能有效地改变当前运行的环境等。后续的计划是取消这种简单的shell命令行直连的功能，而改用类似于SSH登入的功能植入到HTML页面的方式来实现相应的功能。也就是说，使用者仍然需要一份账户和密码来登入服务器，只不过无需下载SSH登录的工具即可在网页上完成相应操作。

本设计在进程间通信的设计中表现得不够“优雅”。本设计使用在cache文件夹中存放代表指令的信标文件的方式来实现信息在进程间的传递，只是一个代行之法，效率并不高。之后的工作中将会寻找一种方式，在php进程和python进程之间进行内存共享，或者通过其他方式进行进程间的通信。

# 参考文献

[1]<https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/>

[2]Brodtkorb A R, Dyken C, Hagen T R, et al. State-of-the-art in heterogeneous computing[J]. Scientific Programming, 2010, 18(1): 1-33.

[3]Caulfield A M, Chung E S, Putnam A, et al. A cloud-scale acceleration architecture[C]//2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016: 1-13.

[4]Robert H N. Theory of the backpropagation neural network[J]. Proc. 1989 IEEE IJCNN, 1989, 1: 593-605.

[5]Goodfellow I, Bengio Y, Courville A. Deep learning[M]. MIT press, 2016.

[6]Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., & Schaa, D. (2012). Heterogeneous computing with openCL: revised openCL 1. Newnes.

[7]王风盼. 基于深度学习的手写数字识别方法研究[D].重庆大学,2018.

[8]鲍云峰. 基于异构计算系统的卷积神经网络应用研究[D].湖北大学,2018.

[9]刘辰雨. 基于卷积神经网络的手写数字识别研究与设计[D].成都理工大学,2018.

[10]李倩倩. 基于深度学习的手写体数字序列的识别研究[D].哈尔滨工程大学,2018.

[11]Wu L, Zhu Z. Towards understanding generalization of deep learning: Perspective of loss landscapes[J]. arXiv preprint arXiv:1706.10239, 2017.

# 致谢

自2016年进入复旦大学以来，学院平易近人的老师和出类拔萃的同学在学业上给予了我很多鼓励和帮助，使我感到四年的光阴没有虚度。在此我必须对陈更生、薛晓勇、叶凡等几位老师表达诚挚的敬意，需要对丁路昶、石广勇、黄智泽、高朋等提供过帮助的学长表示由衷的谢意，也应对易晓玲、王依婷、林昱凯、林山青、方晋北等同学表示衷心的感谢。

时光的指针不断自增，转眼间指向了2020年，世界都充满了未知和迷茫的不定数。新冠病毒的疫情最开始在我的家乡武汉蔓延的时候，我感到前所未有的恐惧。我要感谢那些在一线积极抗疫的医护工作者和党员，感谢他们无私奉献的精神和行动，为人民保住了家园净土，保住了和平安定，扫除了人们心中恐惧的阴霾。一波未平一波又起，世界或许又将迎来百年未有之大变局。半导体行业正处于风口浪尖，我愿和这样的人一起守护家国，又岂有退缩的道理。

我还要感谢我的父母，尤其是我的母亲。我得感谢他们对我投入的精力和感情。我所收获的一切，今后都要加倍返还给他们。今年的一月至五月，由于疫情的原因，我久违地在家生活了四个月。父母都是上前线防疫的公务员，公务十分繁忙。这些日子里，我的母亲在繁忙的公务之间抽出精力安排我的生活用度，我感受到了她对这个家的爱与付出。

我还需要感谢我现在的室友，田益、郑凯祥、蒋凯帆三位同学。寝室氛围融洽是很难得的。我们寝室里很少闹矛盾，在长时间的相处后仍能保持这一点，则尤为可贵。经历岁月的印痕，寝室已经从一个旅馆一样的集散地，变成有温度的地方。

最后，我要感谢我一生中遇到过的所有人，无论是好的、坏的还是平平无奇的。世界是一所大学，所有人都是我在这所大学中的同学，所有经历都是这所大学所开设的课程，而每个人都可以是别人的老师。