卷积神经网络训练和推理的高算力实现方法研究和实现

李天凌 何宾 郑阳扬

（北京化工大学信息科学与技术学院，北京，100029）

## 摘要

算法和算力始终是推动人工智能向前发展的两大动力。平台的算力对于算法的实现成本、性能、功耗和灵活性有着非常重要的影响。目前，人工智能算法模型的训练主要采用高性能GPU平台，人工智能算法模型的推理可使用GPU、CPU和FPGA实现。一方面，由于GPU功耗大、成本高，故不适用于对功耗和成本敏感的应用场景；另一方面，由于神经网络的训练和推理分别采用了不同的算力平台，因此需要在不同算力的平台传输神经网络模型数据，这样就影响了网络的数据处理能力，并且对神经网络的实时性和灵活性造成了影响。

本文围绕人工智能中卷积神经网络训练和推理一体化的高算力实现方法进行研究，提出将卷积神经网络（Convolutional Neural Network，CNN）训练和推理的过程采用以现场可编程门阵列（Field Programmable Gate Array，FPGA）为核心的高性能异构架构（Hybrid Architecture，HA）器件实现。将CNN训练与推理过程中大量且重复的乘法与累加运算用可编程逻辑（Programmable Logic，PL）实现，显著提升CNN训练与推理速度，并降低了整体功耗，为神经网络在对功耗、成本和面积敏感的应用领域提供了一个新的实现方法。

本研究中，首先以包含CNN训练和推理过程的数据流为基础，研究了将训练和推理数据流进行合并的方法；其次，研究使用高级语言对该合并的数据流结构进行高层次描述，并通过高级综合工具（High Level Synthesis，HLS）将高层次描述转换为硬件寄存器传输级（Register Transfer Level，RTL）描述的方法，并生成知识产权核（Intellectual Property，IP）；然后，在HA器件中通过片内AXI总线接口，将用于实现总体控制、数据预处理以及结果分析的PS和该IP核进行连接；最后，通过Xilinx HA 器件对该一体化实现方法进行了测试和验证，测试时采用了MNIST手写体数字验证集。

根据测试结果可知，与GPU相比，在HA器件PL中实现模型训练的收敛速度相同，且训练时间仅为其78.04%。HA器件对本文中提到的两种卷积神经网络的单帧图片处理时间仅为3.31ms与0.65ms，平均识别准确率为95.697%，且总体功耗仅3.22 W@100MHz，适合部署于功耗受限的轻量化领域。

关键词：训练加速、MPSoC、FPGA、卷积神经网络

## 1 引言

近年来，卷积神经网络广泛应用于语音识别、图像分类、图像处理、加速器、解调器等方面，尤其是在图像分类方面表现出巨大的优势[1-2]。目前卷积神经网络算法多样，但由于其运算量巨大，故传统卷积神经网络的实现大都是基于较为完善的PC端框架，利用GPU训练。但由于GPU灵活性较低、成本与功耗较高限制了部分场景下的应用[3-4]，如难以应用于较小移动设备、天文设备等[5]，故而体积小巧的FPGA映入研究人员的眼帘，卷积神经网网络中的卷积运算包含了大量的乘法和加法运算，传统硬件系统实现包含许多串行计算；而FPGA在逻辑单元充足的情况下可以实现全部并行运算，使运用FPGA处理相关数据时的运算速度得到极大的提升[6-7]。相比GPU拥有的数据并行操作而言，FPGA在数据并行基础上加入了流水并行。所以越来越多的研究人员开始使用FPGA来实现CNN算法。

随着对高算力需求的增加，FPGA越来越多地被应用于人工智能模型的推理实现中，并且对神经网络的前向推理起到加速作用并显著降低了设备功耗，但纯FPGA实现CNN的硬件加速方案不可避免地造成了训练与推理相分离的状况，一定程度上降低了神经网络的灵活性与适应性，不能实现对神经网络训练过程中地动态控制。近年来，以FPGA为核心的HA器件的出现为人工智能训练和推理的一体化高算力实现提供了一种新的途径。在HA器件内集成了处理器系统（Processing System，PS）和PL。一方面，这种混合结构具有很高的灵活性，可实现对神经网络训练与推理过程中的动态调控；另一方面，此结构具有极强的并行推理的能力。因此，显著提升神经网络训练和推理性能，为人工智能模型的训练和推理提供了一种新的解决办法。

研究人员着手实现基于FPGA的HA SoC实现神经网络的硬件加速与小型化部署，用HA SoC的多核处理器进行训练、可编程逻辑部分实现推理，以此降低功耗、节约成本并提升性能[8-10]。但神经网络的训练相比于前向推理而言乘法和加法的运算计算量更大，故在HA SoC中的PS实现数据预处理与数据流管理等操作可以充分发挥其灵活性特点，在HA SoC中的PL实现神经网络的训练与推理可充分发挥其并行运算的特点，显著提升系统整体性能。图1为本文提出的神经网络训练与推理一体化方法。

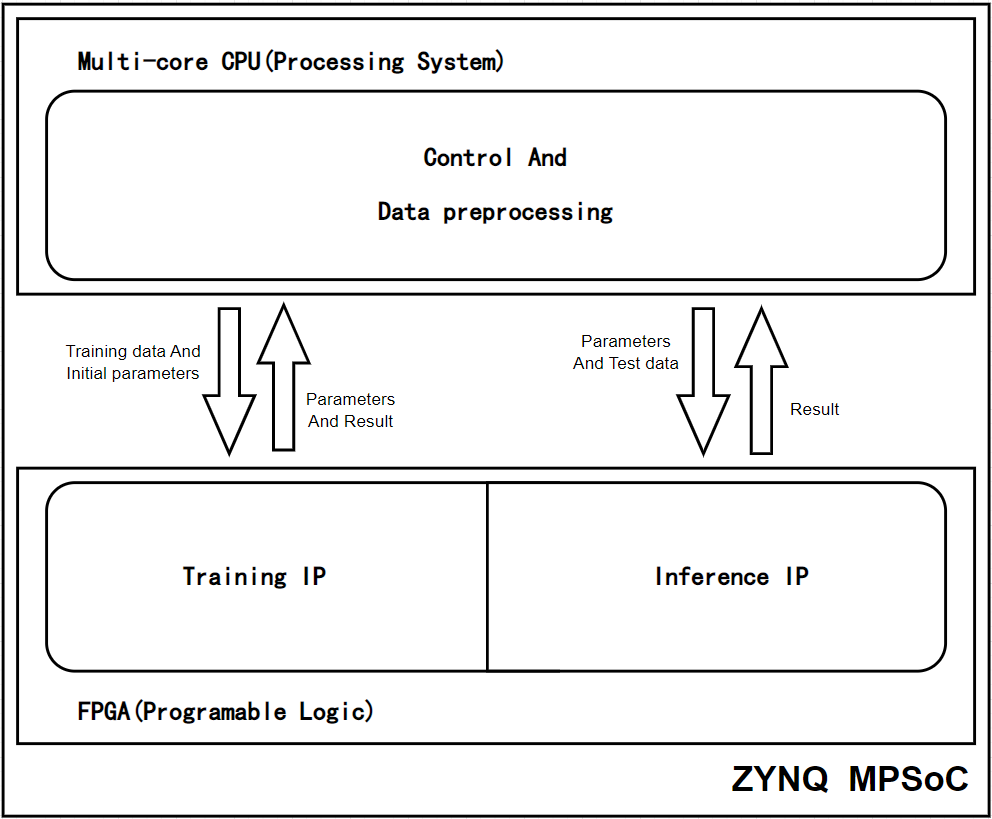


图1 FPGA神经网络训练推理一体化实现方法

综上所述，本文的主要贡献列举如下：

1．利用HA SoC的特点，引入软件和硬件协同设计，平衡实现中的软件和硬件负载，提出了一种全新的神经网络实现方法。充分考虑了软件的灵活性和硬件的并行处理能力，实现了一体化的神经网络训练和推理过程。

2．在该实现中，完全采用PL来实现神经网络的训练和推理过程，而使用PS实现包括数据预处理在内的训练和推理整体过程的调度。

3．根据训练和推理的数据流结构，在PL内根据有限的逻辑资源对流水线和并行性进行了权衡。在保证神经网络前向推理和训练准确率的前提下，尽可能对神经网络的训练过程进行加速。

本文其余的部分的研究内容如下，第二部分介绍了一些相关的前沿性工作。第三部分介绍了LeNet-3.3和LeNet-2.22两种针对MNIST数据集进行识别的卷积神经网络模型和数据流结构，并提出了该神经网络训练和推理在HA SoC上一体化实现的整体设计方案，给出了本设计的架构特点以及软件和硬件负载的分配策略。第四部分详细介绍了上述两种神经网络的高层次描述方法，并对HLS工具转换得到的RTL结构进行了优化，最终生成了包含训练和推理的IP核。第五部分基于HA架构SoC器件构建了用于测试IP核功能和性能的硬件测试环境，并从资源消耗、器件功耗、模型收敛速度、训练时长、推理准确率和推理时长等几个方面对该IP核进行功能测试和性能评估。第六部分根据测试结果分析了IP核的性能，并与目前主流的GPU实现方法进行了比较。最后，总结了论文的贡献并进一步对未来的研究方向做出规划。

## 2 相关工作

卷积神经网络的发展，最早可以追溯到1962年，Hubel和Wiesel对猫大脑中的视觉系统的研究，提出了感受野（Receptive Fields）的概念。1980年，日本科学家福岛邦彦提出了一个包含卷积层、池化层的神经网络结构，他也在上世纪80年就提出过Attention概念和网络。1998年，在这个基础上，Yann Lecun中提出了LeNet-5，将BP算法应用到这个神经网络结构的训练上，就形成了当代卷积神经网络的雏形[11]。

NIST是包含了数字，大写和小写手写字母的大型数据集，MNIST数据集是源于NIST的一个大型的手写字符数据集，包含了从0到9的手写数字图片，具有60000张训练图像和10000张测试图像，图像大小为28 x 28像素，为单通道灰度图像。此数据集常用于训练各种图像处理系统，并广泛应用于深度学习领域[12]。

随着深度学习的发展，卷积神经网（CNN）络逐步成为人工智能领域的成熟算法之一。但CNN的计算复杂度要高于传统算法，网络结构也越来越复杂，越来越多的研究人员寻求CNN的硬件加速方法。Guo K和Sui L提出了Angel-Eye，这是一种可编程且灵活的CNN加速器架构[13]。Angel-Eye是一种将CNN映射到嵌入式FPGA的设计流程，通过FPGA来实现CNN的前向推理过程，从而达到CNN的局部加速。但是，纯FPGA加速器的灵活性明显低于基于FPGA的HA SoC，集成在HA SoC内的多核高性能ARM处理器可完成神经网络的训练过程，搭配HA SoC中的PL加速的前向推理可实现完整的神经网络训练与推理过程。

基于HA SoC的的神经网络部署逐渐成为热门话题，Gschwend D在 Zynq 片上系统上实现一种高效的CNN拓扑结构ZynqNet[14]。ZynqNet是专为图像分类而设计的小型嵌入式神经网络，在PS训练优化过的专属CNN，在PL实现CNN的前向推理加速，使硬件加速CNN精度高于84.5%，计算复杂度仅为5.3亿次乘法累加操作。Zheng（2022）实现了基于Xilinx MPSoC的LeNet-5网络前向推理加速[15]。最终在MNIST数据集和CIFAR-10数据集上的识别准确率分别达到99.5%和75.4%，并且单帧的平均处理时间仅为2.2 ms。诸多研究表明，HA SoC具有更好的灵活性有利于实现和部署神经网络的训练与推理，同时在性能与功耗方面上也优于其他实现方案[16-18]。

综上所述，基于HA SoC的硬件加速让神经网络的推理变得更加灵活，但是很少有研究人员使用PL针对神经网络的训练过程进行加速[20-22]。因此，无法实现模型训练和前向推理的一体化，而一体化将显著提升系统整体性能并且降低系统总成本和功耗[23]。在HA SoC内，将软件的灵活性和硬件的并行处理进行深度融合，PS内运行的软件负责数据预处理并控制数据流，PL执行训练和推理过程中复杂的数据运算，这对于从性能、成本、功耗、灵活性和适应性等方面进一步优化人工智能的算力提供了另一种实现方法[24-25]。

## 3本文方法

### 3.1神经网络训练推理一体化架构

如图2所示，在一体化实现架构中，根据训练和推理的数据流，分配HA SoC中的PS和PL资源。这个架构分为4个部分：（1）在操作系统的调度下，PS负责控制训练和推理的进程、完成训练和推理数据预处理、随机初始化网络权值、更新权值文件，以及对训练和推理结果进行评价；（2）通过内部复用方式，在PL中实现了采用硬件方式的训练和推理过程。（3）通过高级可扩展接口（Advanced eXtended Interface，AXI）接口，将训练数据和初始网络权值加载到PL中，在PL中对模型训练后得到最终网络权值，这些权值保留在PL中用于PL中的推理部分，同时也返回给PS用于对训练效果进行评价。（4）通过AXI接口，将推理所需要的数据加载到PL中，并在PL中执行模型推理，将推理得到的最终结果返回到PS。

图2 系统整体架构

**Image Data**

**(Training)**

**Image Data**

**(Inference)**

**CNN Control**

**Functions**

**Model Parameter File**

**Evaluation**

**Functions**

**AXI Interface**

**Recognition Results**

**Processing System**

**Programable Logic System**

**Hardware Accelerated IP**

**Logics :**

**DSP**

**LUT**

**FF**

**RAMs**

**Training**

**Inference**

**Gradient Backhaul**

**Parameter Updates**

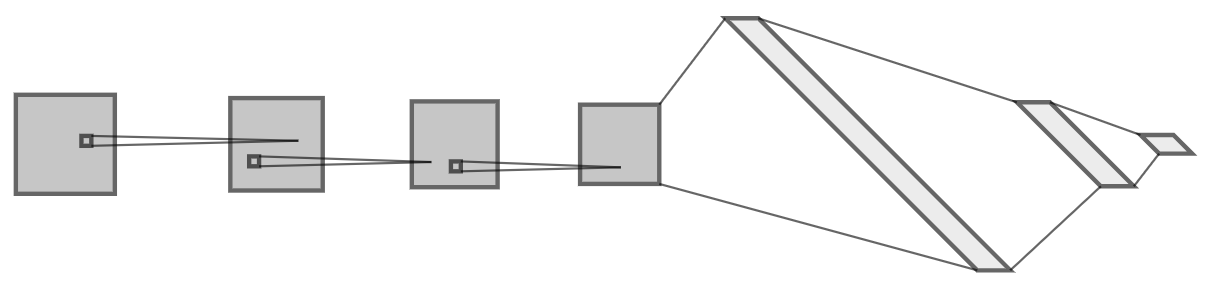
**Conv Layers**

**FC Layers**

### 3.2神经网络的训练与推理实现方法

本设计中使用的训练与推理数据为30\*30像素的单通道灰度图。由于其数据量较少，本设计基于经典卷积神经网络LeNet-5提出了两种改进的网络模型（以下分别称为LeNet-3.3与LeNet-2.22），分析不同网络结构在一体化实现方法中的表现。LeNet-3.3网络去除池化层，增加网络复杂度与图像特征信息量，研究复杂网络在此实现方法中的优势。LeNet-2.22网络采用减少网络权值数量的方法，研究简单网络在该实现方法中的优势。

LeNet-3.3的网络模型如图3所示，该卷积神经网络具有六个网络层，前三层为卷积层，输入为基于修改后MNIST数据集单通道的30x30像素特征图，每个卷积层后取消最大池化层，用于保留所有特征图信息；后面三层为全连接层，其中的最后一层完成图像的分类输出。该卷积神经网络使用改进的ReLU函数作为激活函数。通过softmax函数，对输出层的输出进行归一化处理。



1@30 × 30

1@28 × 28

1@26 × 26

1@24 × 24

1 × 180

1 × 45

1 × 10

Convolution

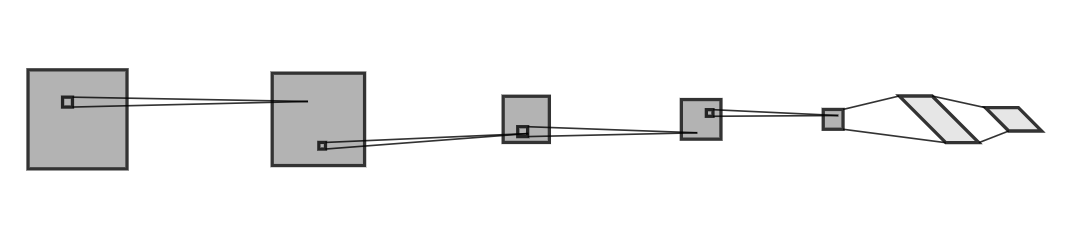
Convolution

Convolution

Full-Connection

图3 LeNet-3.3网络模型

LeNet-2.22的网络模型如图4所示，该卷积神经网络具有六个网络层，前四层为卷积层与池化层交替；后面两层为全连接层，最后一层完成图像的分类输出。该神经网络也使用与LeNet-3.3网络相同的激活函数ReLU和函数softmax。



1@30 × 30

1@28 × 28

1@14 × 14

1@12 × 12

1@6 × 6

1 × 20

1 × 10

Convolution

Convolution

Max-Pool

Max-Pool

Full-Connection

图4 LeNet-2.22网络模型

图5给出了LeNet-3.3网络的图像数据前向传播与梯度损失反向传播的数据流结构。图中，图像数据前向传播用于训练模型，梯度损失反向传播用于模型推理。本质上，神经网络的训练既包含数据前向传播也包含梯度损失的反向传播。因此，PL中实现训练和推理一体化的前提条件是：卷积神经网络的推理要作为模型训练中的一部分。在训练和推理一体化结构中，经过训练得到的网络权值保存在对应的RAM中，在推理时可以直接使用该权值而无需PS的干预。从本质上说，这种训练和推理一体化结构显著减少了算力平台的成本、功耗，同时提高了整个系统的灵活性、可靠性和实时性，为人工智能的训练和推理提供了一个新的解决方法。

图5 网络前向传播与反向传播的数据流程

**mnist\_data**

**[30 × 30]**

**conv\_out\_1**

**[28 × 28]**

**conv\_out\_2**

**[26 × 26]**

**conv\_out\_3**

**[24 × 24]**

**fc\_in\_1**

**[1 × 576]**

**fc\_out\_1**

**[1 × 180]**

**fc\_in\_2**

**[1 × 180]**

**fc\_out\_2**

**[1 × 45]**

**fc\_in\_3**

**[1 × 45]**

**fc\_out\_3**

**[1 × 10]**

**result**

**[1 × 10]**

**conv\_kernel\_1**

**[3 × 3]**

**conv\_kernel\_2**

**[3 × 3]**

**conv\_kernel\_3**

**[3 × 3]**

**fc\_hidden**

**\_layer1**

**[576 × 180]**

**fc\_hidden\_**

**layer2**

**[180 × 45]**

**fc\_hidden\_layer2**

**[45 × 10]**

***Softmax***

**grad\_3**

**wgrad\_3**

**rgrad\_3**

**grad\_2**

**rgrad\_2**

**grad\_1**

**wgrad\_2**

**wgrad\_1**

**kernel\_grad\_3**

**conv\_grad\_2**

**conv\_grad\_1**

**kernel\_grad\_2**

**kernel\_grad\_1**

**conv\_grad\_3**

**rgrad\_1**

**Legend:**

**xxxx**

**Input and output interfaces**

**XXXX**

**Intermediate data**

**XXXX**

**Gradient**

**Data forward propagation path**

**Backpropagation calculates gradients**

**Backpropagation updates parameters**

从平衡HA SoC内的软件和硬件设计资源角度考虑，将卷积神经网络一体化结构进行划分。卷积神经网络的训练和推理涉及到下面的外部操作，包括对图像数据与权值文件的管理、图像数据预处理以及人机交互功能，这些操作可以通过HA SoC内的PS以更加灵活的方式实现。而卷积神经网络本身的训练和推理通过HA SoC中的PL实现，这是因为PL以数据流方式处理数据，因此可以以并行和流水的方式更加高效地执行网络训练与前向推理过程中简单重复且运算量庞大的操作，包括赋值操作、矩阵相乘操作和累加操作等。

在训练模型的过程中，PS可以动态调整学习率、优化器等超参数提高模型的收敛速度、优化模型在推理数据集上的表现；PS可以读取预训练的网络模型权值文件，并将网络权值数据导入PL继续训练，也可以生成随机的初始网络权值，并将初始权值导入PL开始训练。

## 4 核心算法及其实现

本研究通过HLS将C语言描述的神经网络训练和推理模型转换为硬件的RTL描述。与Python语言不同，但C语言中没有提供针对神经网络的库函数，因此就需要使用C语言重新构建神经网络中包含卷积、池化、全连接以及激活等模块的前向传播模型与反向传播模型。

### 4.1 卷积层结构设计

前面提到，本文中的LeNet-3.3卷积神经网络具有六个网络层，主要包括三个卷积层和三个全连接层，因此在设计神经网络加速器时可以针对这两种网络层的结构进行优化。前三个网络层均为卷积层，实现的方法相同。该设计中，将综合考虑前向推理的卷积操作及网络训练中的梯度回传，提高代码的可读性和可维护性。单层卷积前向传播与反向传播路径如图6所示。

图6 数据在卷积层的传播

**Convolutional Layer**

**Forward Propagation**

**Convolutional Layer**

**Error Backpropagation**

edge filling

convolution

convolution

Rotate 180°

convolution

Blue – data Yellow – parameter Green - gradient

（1）卷积层前向传播表示为：

式中， 为第l层卷积的卷积核中对应坐标为(a,b)的元素， 为第l层卷积输出对应坐标为(x,y)的元素。式4-1即为卷积前向传播的关系式，该LeNet-3.3网络的卷积操作不包含偏置、池化以及激活。

卷积层前向传播算法实现如下：

|  |  |
| --- | --- |
| **Algorithm 1: Convolution** | |
| w: Data width  h: Data height  k: Convolutional kernel dimension  input\_matrix: Convolutional layer input data  kernel: Convolution kernel parameters  out\_matrix: Convolutional layer output data | |
|  | 1. for(int i=0;i<w-k+1;i++)   2. for(int j=0;j<h-k+1;j++){  3. out\_matrix[i\*(h-k+1)+j]=0;  4. for(int col=i;col<i+3;col++)  5. for(int row=j;row<j+3;row++)  6. out\_matrix[i\*(h-k+1)+j]+=input\_matrix[col\*h+row]\*kernel[(col-i)\*k+(row-j)];  7. } |

（2）已知第l+1层卷积的输出误差求第l层卷积的输出误差：

式中，误差 是损失函数C对于当前层未激活输出 的导数。在二维卷积中，每一层的 δ 误差都是一个二维矩阵。该矩阵的值表示的是第l层坐标为（x,y）处的delta误差。根据求导的链式法则，式4-2可表示为：

式中坐标 (x',y') 是第l+1层中在前向传播中受第l层坐标 (x,y) 影响到的点。限制条件为：。由 δ 误差的定义可将式4-3化简为：

将卷积前向传播的关系式（式4-1）带入式4-4可得：

根据限制条件与求和与求导的法则上式可化简为：

将限制条件带入式4-6可得：

再令 可得：

由式4-6可知假设已知第l+1层的 δ 误差与第l+1层的卷积核，即可求出第l层的 δ 误差。且对比式4-7与卷积层前向传播（式4-1）可知：第 l 层的 δ 误差为第l+1层的 δ 误差与第l+1层的卷积核旋转180° 的卷积。

但考虑到上述卷积操作中输入与输出矩阵的维度，卷积核维度为k，第l层误差 的维度等同于卷积层输入矩阵 的维度i，第l+1层误差 的维度等同于卷积层输出矩阵 的维度j。由卷积的前向传播可知 j= i-k+1 ，所以在反向传播时需要对第l+1层误差 进行零填充，在其各边填充 (k-1) 使得 维度为j+2\*(k-1)，进行卷积操作后输出的维度为 j+2\*(k-1)-k+1=j+k-1=i，符合第l层误差 的矩阵维度 i 。

综上所述，已知第l+1层卷积的输出误差求第l层卷积的输出误差需要分为三步实现：第一步将卷积 核旋转180°；第二步将第l+1层误差 各边填充 (k-1) 的零值；第三步用上述卷积算法 Convolution 进行运算。旋转卷积核操作实现如下：

|  |  |
| --- | --- |
| **Algorithm 2: OverturnKernel** | |
| k: Convolutional kernel dimension  input\_matrix: Convolution kernel before rotation  out\_matrix: Rotated convolution kernel | |
|  | 1. for(int i=0;i<k;i++)  2. for(int j=0;j<k;j++)  3. output\_matrix[(k-1-i)\*k+(k-1-j)]=input\_matrix[i\*k+j]; |

矩阵零值填充的实现过程如下：

|  |  |
| --- | --- |
| **Algorithm 3: Padding** | |
| w: The width of the matrix to be filled  stride: Each edge is filled with dimensions  input\_matrix: The matrix to be filled  out\_matrix: The filled matrix | |
|  | 1. for(int i=0;i<w+2\*stride;i++)  2. for(int j=0;j<w+2\*stride;j++){  3. if((i>=stride)&&(j>=stride)&&(i<stride+w)&&(j<stride+w))  4. output\_matrix[i\*(w+2\*stride)+j]=input\_matrix[(i-stride)\*w+(j-stride)];  5. else  6. output\_matrix[i\*(w+2\*stride)+j]=0; // 边缘补0  7. } |

（3）已知第l层卷积输出的误差求该层卷积核的误差：

式中 极为第l层卷积核对应元素的误差。带入式4-2可化为：

带入卷积层前向传播（式4-1）可表示为：

简化可得：

对比比式4-11与卷积层前向传播（式4-1）可知：第l层卷积核误差 等于第l-1层卷积输出 与第l层卷积输出误差 的卷积。

综上所述，已知第l层卷积输出的误差求该层卷积核的误差同样可通过上述卷积算法Convolution 实现。

Convolution

Data\_in

Convolution

RAM

FPGA

Convolution

Data\_out

RAM

Convolution kernel

RAM

ap\_auto

Input gradient

RAM

Padding

FPGA

Input gradient

(More)

RAM

Convolution

FPGA

Rotate 180°

FPGA

Output gradient

RAM

Convolution kernel (New)

RAM

Convolution

FPGA

Parameter gradient

RAM

图7 卷积层的前向传播与反向传播实现

将式4-1、式4-7以及式4-11所表示的卷积层操作模型整合后映射为最基本的硬件加速器结构，如图7所示。通过对该基本结构的优化，进一步提高卷积层硬件加速器的整体处理性能。当软件模型通过HLS工具转化为硬件结构时，软件模型的表达方式可能会对最终转化的硬件加速器结构产生不利的影响。在转化为硬件加速器实现时，使用Pipeline命令可以很好地缩短for循环之内的指令触发间隔，从而提高吞吐量和降低时延。

### 4.2 全连接层结构设计

LeNet-3.3卷积神经网络的最后三层均为全连接层，其硬件加速结构相对简单，包括全连接与激活两个过程，下面对其算法的实现模型进行介绍。全连接层的数据通路如图8所示。

**Fully Connected Layer**

**Forward Propagation**

**Fully Connected Layer**

**Error Backpropagation**

multiPLy

multiPLy

multiPLy

transpose

multiPLy

Blue – data Yellow – parameter Green - gradient

activation

transpose

图8 数据在全连接层的传播

#### **4.2.1** 全连接层乘法结构

（1）全连接前向传播表示为：

式中， 为第l层全连接的权值矩阵， 为第l-1层全连接激活后的列向量输出； 为第l层全连接激活前的列向量输出。全连接矩阵乘法的实现如下：

|  |  |
| --- | --- |
| **Algorithm 4: MatrixMultiPLy** | |
| h: Fully connected input vector dimension  h\_out: Fully connected output vector dimension  input\_matrix: Fully connected input vectors  para\_layer：Fully connected weight matrix  out\_matrix: Fully connected output vectors | |
|  | 1. for(int j=0;j<h\_out;j++){  2. output\_matrix[j]=0;  3. for(int i=0;i<h;i++)  4. output\_matrix[j]+=input\_matrix[i]\*para\_layer[i\*h\_out+j];  5. } |

（2）已知第l+1层全连接的输出误差 求第l层激活后的输出误差 ：

将全连接前向传播（式4-12）带入可得：

由矩阵求导规则可得：

其算法实现如下：

|  |  |
| --- | --- |
| **Algorithm 5: CalculateMatrixGrad** | |
| w: Fully connected weight matrix width  h: Fully connected weight matrix height  input\_matrix: Fully connected weight matrix  grad：Fully connected output gradient  out\_matrix: Fully connected input gradient | |
|  | 1. for(int i=0;i<w;i++){  2. output\_matrix[i]=0;//梯度清空，方便累加  3. for(int j=0;j<h;j++)  4. output\_matrix[i]+=input\_matrix[i\*h+j]\*grad[j];  5. } |

（3）已知第l层全连接的输出误差 求第l层全连接权值矩阵的输出误差 ：

将全连接前向传播（式4-12）带入可得：

由矩阵求导规则可得：

其算法实现如下：

|  |  |
| --- | --- |
| **Algorithm 6: MatrixBackPropagationMultiPLy** | |
| w: Fully connected weight matrix width  h: Fully connected weight matrix height  input\_matrix: Fully connected input gradient  grad：Fully connected output gradient  rgrad: Fully connected weight matrix gradient | |
|  | 1. for(int i=0;i<w;i++)  2. for(int j=0;j<h;j++)  3. rgrad[i\*h+j]=input\_matrix[i]\*grad[j]; |

#### **4.2.2** 全连接层激活函数结构

（1）激活函数前向传播表示为：

式中，( ) 表示激活函数， 表示第l层激活函数的输入， 表示第l层激活函数的输出。其中激活函数用LeakyRelu代替Relu避免梯度弥散，算法具体实现过程如下：

|  |  |
| --- | --- |
| **Algorithm 7: Relu** | |
| h: The height of the column vector to be activated  input\_matrix: The column vector to be activated  output\_matrix: Activate the rear column vector | |
|  | 1. for(int j=0;j<h;j++)  2. output\_matrix[j]=max(input\_matrix[j],input\_matrix[j]\*0.05); |

（2）已知第l层激活后的输出误差 求第l层全连接的输出误差 ：

|  |  |
| --- | --- |
| **Algorithm 8: ReluBackPropagation** | |
| h: The height of the column vector to be activated  input\_matrix: Activate the rear column vector  grad: Gradient after activation  output\_matrix: The gradient before activation | |
|  | 1. for(int i=0;i<w;i++)  2. if(input\_matrix[i]>0)  3. output\_matrix[i]=1\*grad[i];  4. else  5. output\_matrix[i]=0.05\*grad[i]; |

对于全连接层，输入与输出的特征图与梯度都是一维向量，权值及其梯度为二维向量。由式4-12与式4-19可知，在实现全连接模块前向传播时，将特征图与权值进行加权运算后，再通过激活函数就得到了全连接模块的输出结果。由式4-15与式4-20可知，在实现全连接模型反向传播时，输入的梯度经过激活函数导数后与权值矩阵的转置进行加权运算后，就得到了反向传播输出的梯度。由式4-18可知，在实现全连接模块更新权值矩阵时，输入的梯度与输入特征图的转置进行加权运算之后，就得到了权值矩阵的梯度，该梯度结合学习率可更新权值矩阵。对式4-12、式4-15、式4-18、式4-19和式4-20进行组合，得到全连接层前向传播与反向传播操作模型，全连接模块的最基本的硬件加速器结构如图9所示。

Fully connected

Data\_in

Matrix multiPLication

RAM

FPGA

Data caching

RAM

Weight matrix

RAM

ap\_auto

Input gradient

RAM

MultiPLication

FPGA

Input gradient

(caching)

RAM

Matrix multiPLication

FPGA

Transpose

FPGA

Output gradient

RAM

Weight matrix

(Transpose)

RAM

MultiPLication

FPGA

Fully connected

Data\_out

RAM

Activate the function

RAM

Derivation

FPGA

Activate the function-Derivation

RAM

Fully connected

Data\_in

(Transpose)

RAM

Matrix multiPLication

FPGA

Weight gradient

RAM

图9 全连接层前向传播与反向传播的实现

使用Unroll命令可以将使用循环表示的全连接层软件模型转化成高效的硬件加速器结构。LeNet-3.3网络对应的硬件IP资源占用详情如表1所示。

表1 LeNet-3.3 IP核资源占用详情

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Training And Test IP** | | | |
| **Total** | **BRAM\_18K** | **DSP48E** | **FF** | **LUT** |
| **419** | **134** | **20962** | **36190** |
| **Utilization(%)** | **96** | **37** | **14** | **51** |

### 4.3 池化层结构设计

池化层夹在连续的卷积层中间，在保留主要特征的同时减少参数和计算量，防止过拟合，提高模型泛化能力。常用的池化操作有最大池化与平均池化，最大池化有助于保留图像的边缘特征，平均池化有助于保留图像的背景特征。本设计中主要针对手写数字进行识别，故采用最大池化。

数据前向传播时，用池化核扫描数据选取该区域内最大值输出，并记录该最大值在原数据中的位置用于反向传播。池化层前向传播算法实现如下：

|  |  |
| --- | --- |
| **Algorithm 9: MaxPool2d** | |
| w: The width of the data before pooling  h: The height of the data before pooling  k: The dimensions of the pooled kernel  input\_matrix: Data before pooling  output\_matrix: Pooled data  locate\_matrix: The position matrix in pooling | |
|  | 1. for(int i=0;i<w/k;i++)  2. for(int j=0;j<h/k;j++){  3. int max\_num=-999;  4. for(int col=i\*k;col<(i+1)\*k;col++)  5. for(int row=j\*k;row<(j+1)\*k;row++)  6. if(input\_matrix[col\*h+row]>max\_num){  7. max\_num=input\_matrix[col\*h+row];  8. locate\_matrix[i\*(h/k)+j]=col\*h+row;  9. }  10. output\_matrix[i\*(h/k)+j]=max\_num;  11. } |

输出的值只取决于该范围内的最大值，故输出的梯度等于该最大元素的梯度，其他元素梯度为0。损失梯度反向传播时，查找其对应位置进行赋值，其他梯度赋值为0。池化层梯度反向传播算法实现如下：

|  |  |
| --- | --- |
| **Algorithm 10: MaxPooBackPropagation** | |
| w: The width of the data before pooling  h: The height of the data before pooling  k: The dimensions of the pooled kernel  input\_matrix: The gradient after pooling  output\_matrix: The gradient before pooling  locate\_matrix: The position matrix in pooling | |
|  | 1. for(int col=0;col<w;col++)  2. for(int low=0;low<h;low++)  3. output\_matrix[col\*h+low]=0;  4. int current\_locate;  5. for(int i=0;i<w/k;i++)  6. for(int j=0;j<h/k;j++){  7. current\_locate = locate\_matrix[i\*(h/k)+j];  8. output\_matrix[current\_locate]=input\_matrix[i\*(h/k)+j];  9. } |

LeNet-2.22中卷积、全连接以及激活操作与上述LeNet-3.3一致。LeNet-2.22网络对应的硬件IP资源占用详情如表2所示。与表1相比，LeNet-2.22加入了卷积层，显著降低了HA SoC器件内部资源的消耗。

表2 LeNet-2.22 IP核资源占用详情

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Training And Test IP** | | | |
| **Total** | **BRAM\_18K** | **DSP48E** | **FF** | **LUT** |
| **27** | **103** | **17443** | **28787** |
| **Utilization(%)** | **2** | **28** | **12** | **40** |

## 5 实验和验证

### 5.1 验证平台的构建

基于Xilinx公司的XCZU3EG-SBVA484 MPSoC，以及片外的2GB LPDDR4存储器，构建了用于验证神经网络训练和推理一体化实现方法的算力平台——Ultra96-V2。在MPSoC内的PS中集成了Arm四核Cortex-A53应用处理单元（APU）和Arm双核Cortex-R5F实时处理单元（RPU），以及在PL中集成了包括7.6MB的块存储器资源 (Block RAM，BRAM)和360个数字信号处理模块DSP48E在内的丰富逻辑资源。

**Zynq Ultrascale+**

**MPSoC**

**Process System**

**Reset**

**Training And Inference**

**AXI Interconnect**

**AXI SmartConnect**

**Reset**

**Reset**

**Reset**

**Reset**

**Clock**

**Clock**

**Clock**

**Clock**

**S\_AXI**

**S\_AXI**

**S\_AXI**

**S\_AXI**

**M\_AXI**

**M\_AXI**

**M\_AXI**

**M\_AXI**

图10 MPSoC内构建的卷积神经网络训练和推理一体化测试结构

如图10所示，基于IP核封装和复用技术，在MPSoC内构建了整个卷积神经网络训练与推理的系统。通过AXI规范，将用于实现网络训练和推理的IP核连接到SoC的PS。各个模块IP核的功能如下：(1) Training And Inference模块用于进行卷积神经网络训练与前向推理的大量计算过程；(2) Zynq Ultrascale+ MPSoC模块是MPSoC内PS的映射；(3) AXI Interconnect模块通过开关结构将多个AXI存储器映射的主设备连接到多个存储器映射的从设备，在该结构中主要作为连接S\_AXI外设的桥；(4) AXI SmartConnect模块用于将AXI外设连接到PS，在该结构中主要作为连接M\_AXI的桥；(5) Process System Reset模块中用于为PS和其他三个模块产生复位信号。

对图10给出的图像识别系统的整体结构进行综合与实现后，两个网络模型所使用PL的逻辑资源如表3所示。

表3 两个网络模型的资源利用率报告

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **LeNet-3.3** | | | | | **LeNet-2.22** | | | | |
| **Utilization Estimates** | **BRAMs** | **DSP** | **FF** | **LUT** | **LUTRAM** | **BRAMs** | **DSP** | **FF** | **LUT** | **LUTRAM** |
| **211.5** | **127** | **20438** | **26199** | **5827** | **13.5** | **97** | **17557** | **17472** | **949** |
| **Utilization (%)** | **97.92** | **35.28** | **14.48** | **37.13** | **20.23** | **6.25** | **26.94** | **12.44** | **24.76** | **3.30** |

由Vivado内集成的功耗分析工具可知，LeNet-3.3图像识别体系统的总功耗仅为2.652W，而用于实现卷积神经网络训练与推理的PL功耗仅为0.387W；LeNet-2.22图像识别体系统的总功耗仅为2.457W，而用于实现卷积神经网络训练与推理的PL功耗仅为0.217W。

### 5.2 验证方法的设计

基于PYNQ框架，使用Python语言编写调用训练和推理IP核的软件代码。在该框架内提供的Overlay编程库为IP核生成可调用的Python API，使得该系统的软硬件协同设计更加便捷。测试数据流如图11所示。

**Training data**

**Test data**

**Model parameters**

**Training And Test**

**Result**

**Data preprocessing**

**Data preprocessing**

**Load**

**Save**

**Load**

**Training**

**Test**

**Multi-core CPU(PS)**

**FPGA(PL)**

**AXI Interface**

**Evaluation analysis**

图11 用于测试卷积神经网络训练和推理硬件结构的数据流

训练过程数据流如下：PS加载网络权值或生成随机初始网络权值，并对训练数据进行预处理，通过AXI接口将初始网络权值与训练数据导入PL。PL采用迭代训练方式更新网络权值，并将训练后的网络权值与训练结果通过AXI接口导出。最终，PS更新权值文件并对训练效果进行评估。

推理过程数据流如下：PS加载网络权值并对推理数据进行预处理，通过AXI接口将其导入PL。PL逐一进行推理，并将推理结果结果通过AXI接口导出。最终，PS对推理结果的准确率与推理时间进行评估。

## 6 实验结果与分析

目前，对神经网络模型的训练主要是在CPU或GPU上完成的，本设计采用的是硬件方法来训练神经网络模型，三类平台的整体性能如表4所示，其中网络训练和推理时长是利用各个硬件平台提供的时间函数在实际运行过程中测量得出，并不包含图像预处理以及导入时间。

纵向对比可知，本文提到的LeNet-2.2网络结构在训练效率方面远远优于LeNet-3.3网络结构，池化层显著减少了网络参数提升了训练效率，所以接下来主要对比各个平台在运行LeNet-2.2网络的性能。

横向对比可知，本设计所用的MPSoC在售价方面确实不太占优势，但是该设备平均功耗仅3.02w是CPU功耗的6.86%，是GPU功耗的4.03%，且MPSoC设备体积小巧更符合轻量化部署的要求。在MPSoC的PL内完成一张图像的训练时间仅为GTX1050训练时间的78.03%，仅为Intel i5-4300U训练时间的55.98%。因此，基于HA SoC中的PL实现对神经网络模型的训练效果并不比逊色于GPU，但其缺点在于牺牲了算法部署的灵活性，开发周期也略长。经过测试集上1000张图片的验证,LeNet-2.22模型在ZYNQ UltraScale+ MPSoC上的单帧图像推理时间为0.65ms，而在Intel i5-4300U CPU与GTX1050 GPU上单帧图像的处理时间分别为2.01ms与0.78ms，推理速度分别平均提升了3.09倍与1.2倍。

表4 整体性能对比

|  |  |  |  |
| --- | --- | --- | --- |
| **设备** | **CPU** | **GPU** | **MPSoC** |
| **型号** | **Intel i5-4300U** | **GTX1050** | **ZYNQ UltraScale+ MPSoC (PL)** |
| **售价** | **X** | **929￥** | **2600￥** |
| **工作频率** | **2.5GHz** | **1455MHz** | **100MHz** |
| **LeNet-3.3平均功耗** | **44W** | **75W** | **3.22W** |
| **LeNet-3.3训练时长/image** | **55.68ms** | **24.6ms** | **19.2ms** |
| **LeNet-3.3推理时长/image** | **10.14ms** | **4.08ms** | **3.31ms** |
| **LeNet-2.22平均功耗** | **44W** | **75W** | **3.02W** |
| **LeNet-2.22训练时长/image** | **1.84ms** | **1.32ms** | **1.03ms** |
| **LeNet-2.22推理时长/image** | **2.01ms** | **0.78ms** | **0.65ms** |

如表5所示，在HA SoC中的PL实现训练的收敛速度较快，与GPU以及CPU上效果基本一致。

表5 卷积神经网络的收敛速度

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **CPU** | | **GPU** | | **MPSoC** | |
| **Epoch** | **Lose** | **Learning rate** | **Lose** | **Learning rate** | **Lose** | **Learning rate** |
| **5** | **0.378564** | **0.0038268049** | **0.444973** | **0.0050369459** | **0.622568** | **0.0089149121** |
| **10** | **0.011166** | **0.0000095806** | **0.039344** | **0.0000815240** | **0.112957** | **0.0004897260** |
| **15** | **0.011789** | **0.0000105072** | **0.027722** | **0.0000449574** | **0.015632** | **0.0000169758** |
| **20** | **0.001402** | **0.0000002816** | **0.008204** | **0.0000056733** | **0.002037** | **0.0000005311** |

如图12所示，最终加速器IP核对MNIST手写体数字体验证集的平均识别准确率为95.697%，准确率与该模型在计算机上的表现一致。下图为部分实验结果截图的展示：

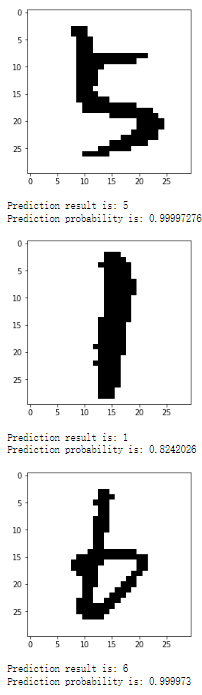
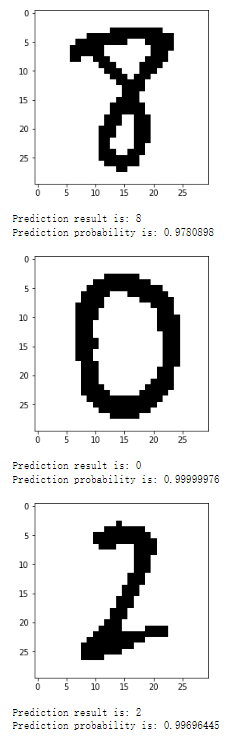
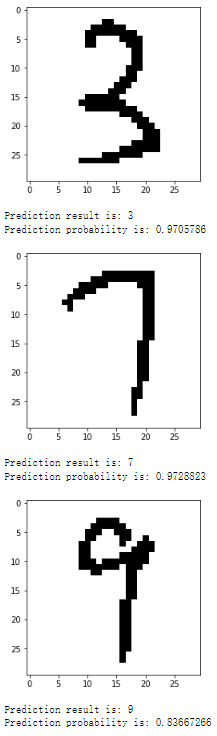


图12 加速器在MNIST数据集上的识别结果

## 7 讨论

在本文中，采用HA SoC（Ultra96-V2）实现网络的训练和推理一体化构建，创新性地在PL实现了神经网络的训练和推理过程，显著降低了HA SoC上多核处理器的运算负载。该实现方法相比于传统的GPU实现，显著降低了设计成本和设备功耗并提升了网络训练与推理速度。最大限度地发挥出了高性能HA SoC上多核处理器的灵活性与可编程逻辑门阵列并行计算的强大算力，为人工智能在对成本、性能和功耗要求比较苛刻的应用场景中，找到了新的解决方案。Zheng（2022）[15]的研究实现了对LeNet-5网络推理的加速，在MNIST数据集和CIFAR10数据集上的识别准确率分别达到了99.5%和75.4%，图片的平均单帧处理时间仅为2.2ms，但是其网络的训练仍是在PS进行的，本文提到的方法在此基础上进一步实现了使用PL对CNN训练的加速，使训练与推理更加整体化，对MNIST手写体数字体验证集的平均识别准确率为95.697%，单张图片训练时长1.03ms，单张图片推理时长0.65ms。

相信随着集成电路技术的不断发展，HA SoC的性能也会越来越高。该器件将逐步成为人工智能的主流实现平台之一，其强大的算力也将一定程度上推动人工智能技术的发展。在之后的研究中，我们将致力于将这种架构应用到更复杂的场景中，并探索更复杂的网络模型在高性能HA SoC上的实现方法。

## 8 结论

首先，本文详细探究了神经网络训练与推理过程的数据流结构，讨论了利用异构架构MPSoC的PL实现网络训练与推理的可能性及其优劣。然后，结合Xilinx最新的多核异构架构设备MPSoC的特点与优势，研究了将训练与推理的网络模型转换为硬件逻辑的一体化实现方案，显著提升了MPSoC设备逻辑资源的利用率。最后，使用Xilinx HA设备对集成实现方法进行测试验证，并使用MNIST手写数字经验证书集进行测试。结果表明，利用MPSoC的PL实现网络训练与推理一体化的方式显著降低了人工智能训练和推理的功耗，为人工智能领域功耗受限的部署场景提供了新的解决方案，由于其整体化设计显著提升了设备的资源利用率，最大程度上降低了处理器的运算负载，平衡了算力与灵活性。但是，此方法目前也有一定的局限性，广泛部署成本略高。相信随着半导体技术的发展以及异构架构SoC设备的普及，此问题也可以迎刃而解。

## 9 参考文献

[1] Y. LeCun,L. Bottou,Y. Bengio,P. Haffner. Gradient-based learning apPLied to document recognition[J] . Proceedings of the IEEE . 1998 (11)

[2] Zhang C , Li P , Sun G , et al. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. ACM, 2015.

[3] Colbert I , Daly J , Kreutz-Delgado K , et al. A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing ApPLications?[J]. 2021.

[4]何宾, 张艳辉. Xilinx FPGA 数字信号处理权威指南 : 从 HDL 到模型和 C 的描述 : The definitive guide of digital signal processing on Xilinx FPGA from HDL to model and C description[M]. 清华大学出版社, 2014.

[5]戴源,白雨鑫,张伟,陈鑫.基于 Vivado HLS 的硬件设计效能评估[J].电脑知识与技 术,2021,17(19):1-4.

[6] S. I. Venieris and C. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 40-47, doi: 10.1109/FCCM.2016.22.

[7] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor and S. Areibi, "Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks," 2016 International Conference on Field-Programmable Technology (FPT), 2016, pp. 265-268, doi: 10.1109/FPT.2016.7929549.

[8] 华士豪.基于 FPGA 的轻权重手写数字系统设计优化[J].电子制作,2020(16):6-7+37.

[9] Y. A. Bachtiar and T. Adiono, "Convolutional Neural Network and Maxpooling Architecture on Zynq SoC FPGA," 2019 International Symposium on Electronics and Smart Devices (ISESD), 2019, pp. 1-5, doi: 10.1109/ISESD.2019.8909510.

[10] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," 2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), 2016, pp. 1-6, doi: 10.1109/ICSPIS.2016.7869873.

[11] Y. LeCun,L. Bottou,Y. Bengio,P. Haffner. Gradient-based learning apPLied to document recognition[J] . Proceedings of the IEEE . 1998 (11)

[12] Cohen G, Afshar S, TaPSon J, et al. EMNIST: Extending MNIST to handwritten letters[C]//2017 international joint conference on neural networks (IJCNN). IEEE, 2017: 2921-2926.

[13] Guo K, Sui L, Qiu J, et al. Angel-eye: A comPLete design flow for mapping CNN onto embedded FPGA[J]. IEEE transactions on computer-aided design of integrated circuits and systems, 2017, 37(1): 35-47.

[14] Gschwend D. Zynqnet: An fpga-accelerated embedded convolutional neural network[J]. arXiv preprint arXiv:2005.06892, 2020.

[15] Zheng Y, He B, Li T. Research on the Lightweight DePLoyment Method of Integration of Training and Inference in Artificial Intelligence[J]. ApPLied Sciences, 2022, 12(13): 6616.

[16] 王巍, 周凯利, 王伊昌, 等. 卷积神经网络 (CNN) 算法的 FPGA 并行结构设计[J]. 微电子学与计算机, 2019, 36(4): 57-62.

[17] 卢冶, 陈瑶, 李涛, 等. 面向边缘计算的嵌入式 FPGA 卷积神经网络构建方法[J]. 计算机研究与发展, 2018, 55(3): 551-562.

[18] Wu D, Zhang Y, Jia X, et al. A high-performance CNN processor based on FPGA for MobileNets[C]//2019 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2019: 136-143.

[19] Bai L, Zhao Y, Huang X. A CNN accelerator on FPGA using depthwise separable convolution[J]. IEEE Transactions on Circuits and Systems II: Express Briefs, 2018, 65(10): 1415-1419.

[20] Nguyen D T, Nguyen T N, Kim H, et al. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019, 27(8): 1861-1873.

[21] Liu B, Zou D, Feng L, et al. An FPGA-based CNN accelerator integrating depthwise separable convolution[J]. Electronics, 2019, 8(3): 281.

[22] Geng T, Wang T, Sanaullah A, et al. FPDeep: Acceleration and load balancing of CNN training on FPGA clusters[C]//2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018: 81-84.

[23] Lentaris G, Stratakos I, Stamoulias I, et al. High-performance vision-based navigation on SoC FPGA for spacecraft proximity operations[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2019, 30(4): 1188-1202.

[24] Bai L, Zhao Y, Huang X. A CNN accelerator on FPGA using depthwise separable convolution[J]. IEEE Transactions on Circuits and Systems II: Express Briefs, 2018, 65(10): 1415-1419.

[25] Ma Z, Ding Y, Wen S, et al. Shoe-print image retrieval with multi-part weighted cnn[J]. IEEE Access, 2019, 7: 59728-59736.