Hindawi
Mobile Information Systems
Volume 2022, Article ID 8021288, 15 pages
https://doi.org/10.1155/2022/8021288

Research Article

FPGA-Based Real-Time Embedded Fish Embryo Detection System

Mengqi Wang,[1] **Guofu Feng**,[1] **Ming Chen**,[1] **Ruijuan Ye**,[1] **and Yaohui Wang**[2]

[1] *Key Laboratory of Fisheries Information, Ministry of Agriculture and Rural Affairs, Shanghai Ocean University, Hucheng Ring Road 999, Shanghai 201306, China*
[2] *Nantong Long Yang Aquatic Company, Haian 226600, Jiangsu, China*

Correspondence should be addressed to Guofu Feng; gffeng@shou.edu.cn

Received 17 August 2022; Revised 6 September 2022; Accepted 19 September 2022; Published 7 October 2022

Academic Editor: Jiafu Su

In today's aquaculture industry, the period of fish embryos needs to be detected in real time because different periods of fish embryos require different environments for their cultivation. The paper proposes an FPGA-based real-time embedded fish embryo detection system to solve the problems of today's fish embryo period detection, which consumes a lot of human resources and has low accuracy. Based on the selection of YOLOv3 as the basic model, this paper combines model optimization and hardware acceleration to make the whole system achieve the effect of real-time detection. With only a 4.23% reduction in the mAP value of the model, the inference speed of the model was improved by nearly 150%. Finally, we achieved 34.1 fps of real-time fish embryo detection on a 100 MHz FPGA chip, while having 82.49% mAP, and the whole embedded system only consumed 2.524w while running. The results show that the FPGA-based real-time fish embryo detection system proposed in this paper is fully capable of meeting the needs of today's aquaculture embedded devices. It is beneficial to promote the transformation of fish embryo analysis from traditional manual methods to AI-based embedded devices and facilitate the rapid analysis of fish embryos.

1. Introduction

Fish embryonic development has always been an extremely critical aspect of aquaculture. It is crucial for the continuation of rare fish species, high yield breeding, and sustainable development of fishery resources [1]. Since a mismatched culture environment can have a significant impact on embryo development, mainly in terms of survival rate and malformation rate, real-time fish embryo detection results are needed to adjust the corresponding culture environment [2–4]. Because fish embryos are very small before they are fully developed, the traditional method is to manually operate a microscope to observe the growth of fish embryos [1]. The approach has many problems, such as the difficulty of deploying equipment in fish culture greenhouses, the experience gap between analysts that affects the accuracy of identification, and the difficulty of analyzing directly in the culture environment. Therefore, automation and miniaturization of equipment are the main research directions for fish embryo detection in the future [5, 6].

It is difficult to achieve good results in aquaculture by relying only on embryo analysis by human eyes, so it is clearly not feasible to spend a lot of human resources to perform embryo analysis. Artificial intelligence would be a good solution to solve the problem in traditional methods. In 2017, Ishaq et al. demonstrate the potential of a deep learning approach for accurate high-throughput classification of whole-body zebrafish [7]. In 2019, Rauf et al. develop a framework based on improved CNN to classify the fish species [8]. In 2021, Naderi et al. develop CNN-based framework for cardiac function assessment in embryonic zebrafish from heart beating videos [9]. With the widespread use of artificial intelligence in cytology, bioinformatics, and medicine [10–12], embryo detection in aquaculture and academia is gradually experimenting with the approach. Qian et al. [13] established a microinjection system for detecting zebrafish embryos using CNN. However, the method is more suitable for deployment in the laboratory and cannot be directly applied to aquaculture. Xu et al. [14] built a Drosophila embryo detection framework using multifeature fusion (MFF) CNN on GPU. Dirvanauskas et al. [15] used CNN to classify embryonic development. The above studies are CPU based for both classification and detection tasks for CNNs and use GPUs for acceleration. In environments such as greenhouses for aquaculture, the approach may make it difficult to deploy equipment directly to the field due to various uncertainties. The conventional CPU + GPU architecture is difficult to fully meet the needs of aquaculture due to various instrument failures in environments such as greenhouses due to temperature and humidity.

Recently, field programmable gate arrays (FPGAs) have attracted increasing attention from researchers because of their excellent performance, high energy efficiency, fast development cycles, and strong ability of refactoring [16–18]. Although GPUs outperform FPGAs in terms of inference efficiency, dedicated devices made of FPGAs significantly outperform GPUs in terms of power consumption. Such characteristics make FPGAs a better choice for embedded devices [19–21]. In addition, FPGAs can design camera devices directly into the programable logic (PL), allowing FPGAs to process video streams directly with near-no latency [22]. And a series of modern chips represented by ZYNQ make up for the weak logic processing power by embedding ARM processors in FPGAs, fully meeting the needs of deep learning in terms of flexibility [17, 23].

2. Related Work

To the best of our knowledge, we are not aware of any implementation of embryo detection on FPGA.

Nowadays, there are indeed many researchers who have achieved some results in the intersection of FPGA and other disciplines. Zhao et al. [24] built an FPGA-based underwater image recognition system. The system successfully deploys a convolutional neural network on Xilinx's ZYNQ7035 chip. Wei et al. [25] complete an FPGA-based remote sensing image classification system. Luo and Chen [26] built an FPGA-based defect detection system for additive manufacturing, and they used a binarized neural network (BNN) to turn the computational part of the neural network into the most suitable FPGA bit operation. Al Koutayni et al. [27] built the first CNN-based gesture recognition system on ZYNQ. However, the above approach does not take into account the optimization of the model itself, and since the system itself only deals with simple classification problems, it is only necessary to deploy the model to the FPGA. However, if the model structure or the data type of the model can be changed to directly reduce the computation of the model itself, the inference speed of the model will be significantly improved.

To solve the above problems, we propose an FPGA-based embedded system to achieve real-time detection of fish embryos. The embedded system uses a combination of hardware and software to improve the inference speed. We first performed channel pruning, layer pruning, and INT4 quantize on the pretrained model. After pruning, the model is retrained to fine-tune it. In quantization, the TQT method is used to increase the inference speed of the model with as little loss of accuracy as possible. After completing the optimization of the model, we designed an efficient hardware architecture on ZYNQ. The architecture uses FPGAs to implement the computationally intensive part and an ARM CPU to implement the logic control part. By deploying the optimized model to this structure, we have accomplished the detection of fish embryos in real time.

3. Software Design Process

3.1. YOLOv3-Based Fish Embryo Detection. The target detection model chosen for the paper is the YOLOv3 model proposed by Redmon [28]. YOLOv3 was chosen as the final model for our implementation of embryo detection. Because compared to YOLOv4 and YOLOv5, the accuracy of YOLOv3 is not as good as these two models, but YOLOv3 gets the simplicity of the network model and higher inference speed with the cost of some accuracy on the data set relevant to this application. For embryo detection tasks with a small number of classifications and large of image similarity, the accuracy of YOLOv3 is also fully capable of handling the corresponding needs. Faster R-CNN has higher accuracy but slower inference speed. MobilenetV3 has faster inference speed but poorer performance for small target detection. The YOLO series, which combines inference speed and accuracy, is more suitable for deployment on FPGA platforms to accomplish the goal of real-time embryo detection than the previously mentioned models [22, 29, 30].

Figure 1 illustrates the network structure of YOLOv3. YOLOv3 adds a BN layer and a LeakyReLU layer [31] after each convolutional layer in the network. This structure is called DBL (DarknetConv2D + BN + LeakyReLU). The DBL structure is the basic component of the YOLOv3 network. As seen in Figure 1, two DBLs form the Res structure by direct connection and hop-level connection. Five res structures and one DBL structure form the backbone of YOLOv3. The input image size of YOLOv3 is $416 \times 416 \times 3$. There are three outputs in traditional YOLOv3 with feature map depths of 255 and an image edge ratio of 13 : 26 : 52. The depth of each output feature map is based on the bounding box of the YOLO algorithm.

3.2. YOLOv3 Optimization Strategy for FPGAs. Since deploying the entire YOLOv3 model directly to the FPGA would put a huge strain on the memory space and computational resources of the entire system. YOLOv3 has 52 convolutional layers. Direct deployment without any optimization of the model will make the network inference time much higher than the expected time. Thus, in the paper, model pruning and model quantization of YOLOv3 are chosen to reduce the calculation of the network. The process of YOLOv3 model optimization is given in Figure 2. Following the training of the initial neural network, the model will be pruned. After the pruning of the model, the accuracy lost by the pruned model is compensated by fine-tuning operations. In this paper, we choose to use the original data set to retrain the pruned model again. Then, fine-tuning operations are performed to compensate for the lost

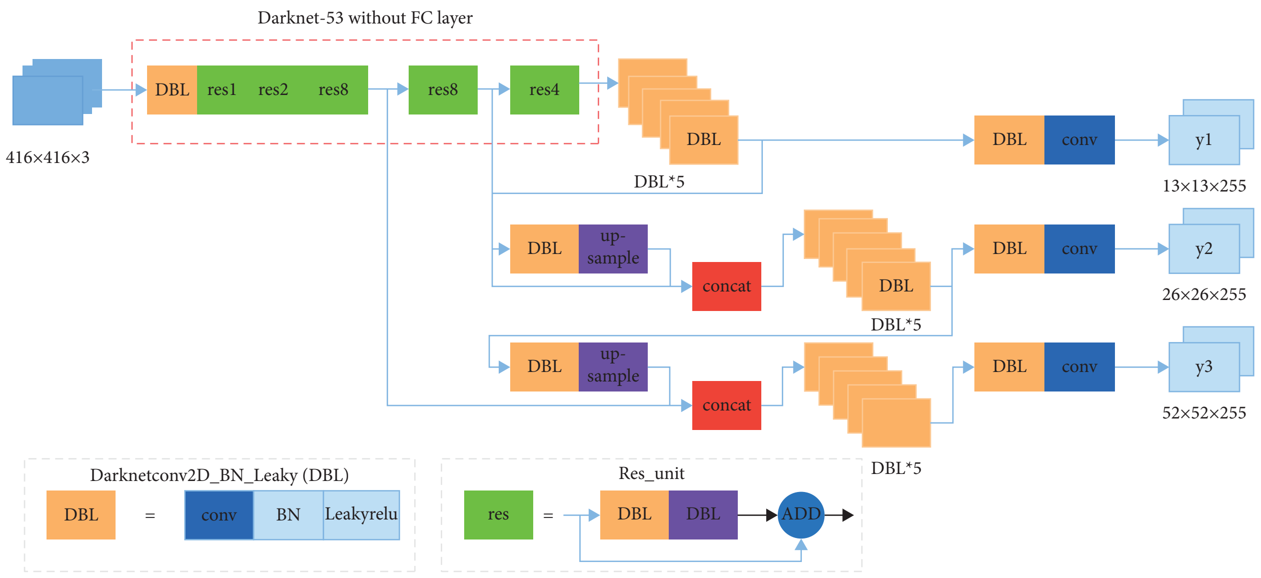


Figure 1: Network structure of YOLOv3.

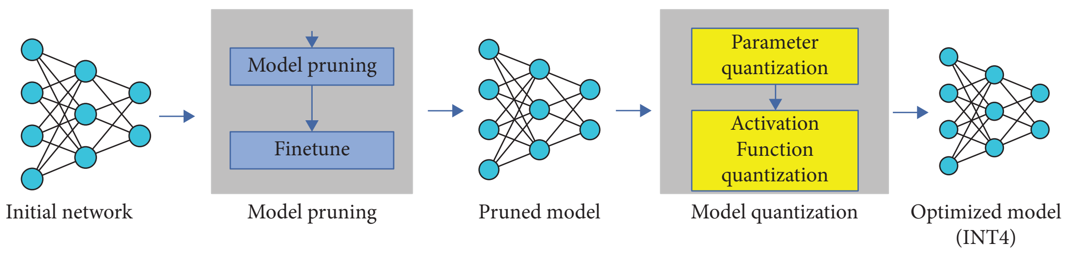


Figure 2: Model optimization process.

accuracy of the model after pruning. The approach taken in this paper is to use the original data set to train the pruned model again. Finally, the whole model is quantified into the INT4 model. The quantized YOLOv3 model will be more suitable for computing on FPGAs.

3.2.1. Pruning of YOLOv3. The pruning operation for convolutional neural networks is classified into layer pruning and channel pruning. The pruning of the network model can reduce the computation of the model and improve the inference speed of the model. Traditional pruning is mostly based on the size of the model's weights. This is not suitable for YOLOv3, which has an extremely large number of weight parameters. In order to achieve better effect, a pruning method based on BN layer scaling factor is used in this paper. Since more than 90% of the operations of the convolutional neural network are done in the convolutional layer, the importance of the convolutional layer in the network should be considered when pruning the model. In YOLOv3, the basic unit is the DBL structure, in which the batch normalization (BN) layer plays the role of accelerating the convergence of the network and controlling the occurrence of the overfitting phenomenon. In the DBL structure, the BN layer is after the convolution layer and before the activation function LeakyReLU. Its calculation formula is as follows:

$$x_{\text{out}} = \frac{\gamma\left(x_{conv} - \mu\right)}{\sqrt{\sigma^2} + 0.000001} + \beta. \tag{1}$$

In formula (1), μ is the mean value of all input x in an input batch; γ is called the scaling factor; σ^2 is the variance; β is the bias; and $x_{\text{conv in channel } c}$ is the computed result of the convolution layer output in channel c of the input, which can be expressed as

$$x_{\text{conv in channel } c} = \sum_{i=0}^{n} \left(x_{ic} * w_{ic}\right). \tag{2}$$

`In equation (2), w_{ic} is the weight required in the convolution operation and is stored in the corresponding weight file. As seen from equation (1), the result of the convolutional layer outputs is directly affected the scaling factor γ, and the magnitude of γ can be understood as the importance of the convolutional layer output result in the next round of operations. The larger the scaling factor γ is, the more important its corresponding channel is in the overall DBL structure. In order to make the distribution of the scaling factor γ in the DBL structure more suitable for the pruning operation, we add the scaling factor γ to the loss function of the model and perform sparse training again on the already trained model. The formula is as follows:

$$\text{Loss} = \text{Loss}(w) + \lambda \sum_{\gamma \in \Gamma} \text{Reg}_g(\gamma). \tag{3}$$

In formula (3), $\text{Loss}(w)$ is the original loss function of YOLOv3; $\text{Reg}_g(\gamma)$ is the sparse regular term; Γ is the set of γ; and λ is the sparsity factor. In this paper, stochastic gradient descent (SGD) optimization is used, and the gradient calculation process can be written as

$$\nabla \gamma^* = \nabla_\gamma \lambda \sum_{\gamma \in \Gamma} \text{Reg}_g(\gamma). \tag{4}$$

Assuming that the learning rate is η, the parameters in equation (4) can be updated as follows:

$$\gamma' = \gamma - \eta \nabla \gamma^*, = \gamma - \eta \lambda \nabla_\gamma \sum_{\gamma \in \Gamma} \text{Reg}_g(\gamma). \tag{5}$$

From equation (5), it is clear that calculating the amount of variation of the γ parameter only requires calculating the partial derivatives of the regular terms and multiplying them by the corresponding sparsity factor $\eta\lambda$ throughout the training process of the network. After the sparse training, channels with relatively small absolute values of γ, i.e., channels with γ parameters close to 0, are removed to perform channel pruning on the model. In this paper, channels with scaling factor γ less than 10^{-3} are removed.

Figure 3 shows the specific operation of channel pruning. In the convolution operation from layer j to layer k, channel 2 and channel 4 in the convolution layer j will be removed because their value of the scaling factor γ is smaller to other channels and less than 10^{-3}.

After the channel pruning, the YOLOv3 still needs to be layer pruned due to insufficient compression of the model. Since the smallest unit in the YOLOv3 structure is the DBL structure, the value of all γ on the BN layer can also reflect the importance of the DBL structure in the whole network. The γ of all remaining channels in each layer is averaged, and the average value is used as a criterion to rank the calculation importance of all DBL structures at the time of deletion. A pruning rate will be set to remove unimportant DBL structures in the network. As shown in Figure 4, layer pruning was performed to cull the model DBL structure based on the mean value of γ. DBL2 and DBLM with smaller mean values of γ were removed.

After the whole pruning operation is completed, a narrower, more compact model is obtained, which needs to be fine-tuned since the accuracy of this model will necessarily decrease. In this paper, the original data set is used as the fine-tuned data set to train the pruned model for another 50 epochs with the original loss function and 1% learning rate. After fine-tuning, the pruned model will be quantified.

3.2.2. Quantification of the Pruned YOLOv3. Traditional quantization methods used in FPGAs generally choose to convert the entire floating-point model into an 8 bit fixed-point model (INT8). This type of quantification method generally discards the decimal values directly in the quantification process [32]. However, such an approach leads to a significant decrease in the accuracy of the network model. To minimize the loss of model accuracy due to quantization while increasing the speed of inference as much as possible, the trained quantization threshold (TQT) method is used in this paper to transform the already pruned single-precision floating-point model (FP32) into a 4 bit fixed-point model (INT4) during the training process [33], where the quantization function $q(x; s)$ can be written as

$$q(x; s) = \text{clip}\left(\text{p}\left\lfloor \frac{x}{s} \right\rfloor; n, p\right) \cdot s. \tag{6}$$

In formula (6), x is the value to be quantified, and s is called the quantization scale factor. When x is a signed number, $n = -2^{b-1}$, $p = 2^{b-1} - 1$, and $s = 2^{\lceil \log_2 t \rceil} / 2^{b-1}$. b is the bit width of quantification results. t is a trainable parameter, which is used as a quantization threshold in equation (6). It is usually set to the maximum of the weights. $clip\ (a; b, c)$ function is used to restrict a to the middle of b, c. Let a equal c when a is greater than c, and let a equal b when a is less than b, $\lfloor x \rceil$, i.e., rounding the input x. Since the quantization scale factor s is a power of 2, it is more friendly to hardware, and the pruning operation in the previous section has removed some of the outliers and promoted a tighter distribution of weights and activation functions, which makes the quantized results more accurate.

As mentioned before, the threshold t is a learnable parameter in training, and $\log_2 t$ is used to control the range of the parameter after quantization. The gradient of t can be obtained by the chain rule. The gradient is calculated as follows:

$$\nabla_{(\log_2 t)} q(x; s) := s \cdot \ln 2 \cdot \begin{cases} \left\lceil \frac{x}{s} \right\rfloor - \frac{x}{s}, & n \leq \left\lceil \frac{x}{s} \right\rfloor \leq p, \\ n, & \left\lceil \frac{x}{s} \right\rfloor < n, \\ p, & \left\lceil \frac{x}{s} \right\rfloor > p. \end{cases} \tag{7}$$

Then, the gradient of x can be obtained as

$$\nabla_x q(x; s) := \begin{cases} 1, & n \leq \left\lceil \frac{x}{s} \right\rfloor \leq p, \\ 0, & \text{otherwise}, \end{cases} \tag{8}$$

where we define the gradient of the integrable function $\lceil x \rfloor$ as

$$\frac{\partial}{\partial x} \lceil x \rfloor = \frac{\partial}{\partial x} \lceil x \rfloor = 1. \tag{9}$$

In this paper, Algorithm 1 is used to quantify the entire network during the quantization training.

4. Hardware Design Process

4.1. General Architecture. In the paper, a heterogeneous architecture is used to implement the whole system. The architecture can be divided into processing system (PS) and programable logic (PL). The PS part consists of a CPU that can be used as a controller and an off-chip memory that is

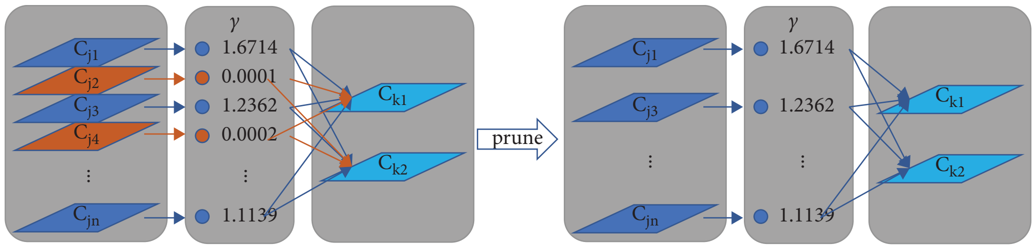


FIGURE 3: Channel pruning.

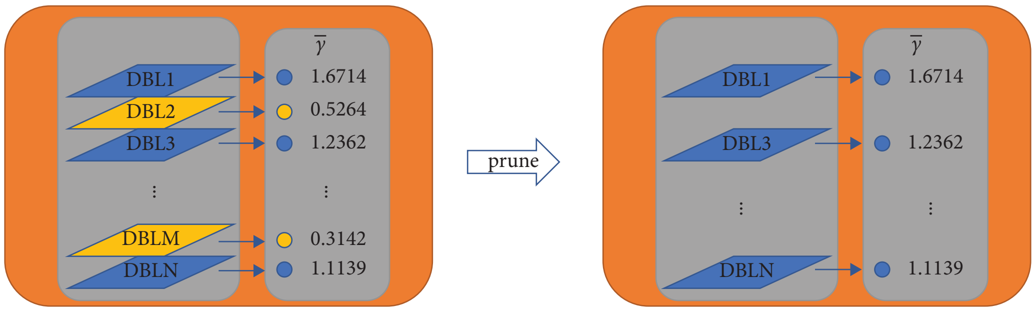


FIGURE 4: Layer pruning.

```
    Full precision inputs, weights and bias, outputs: $X, W$, bias
    Bit width: for inputs and weights, $b = 4$, for bias, $b = 8$
    Output: $Y$, predict number: Label
(1) Calculate q(x), q(w), q(bias) according to equation (6)
(2) $Y$ = Forward(q(x), q(w), q(bias))
(3) Calculate losses and regularize all learnable parameters: $L$ = loss ($Y$, Label)
(4) According to Equation (8), calculate gradient: $\nabla_X L = \partial L/\partial q(x) * \partial q(x)/\partial X, \partial q(x)/\partial X$
(5) Use the Adam optimizer [31] to update a portion of the full precision parameters according to Equation (9); $W$ = Backward (W,
    Adam, $\nabla_X L$)
(6) Returning to 1.
```

ALGORITHM 1: Quantization algorithm for each layer.

used to store data; the PS is responsible for the scheduling and storage of the entire system. The PL part consists of several custom IP cores implemented on FPGAs, each of which is responsible for implementing a network layer function; the PL part is responsible for the computational part of the system. An entire system is implemented on Xilinx ZYNQ chip. Figure 5 illustrates the overall architecture of the system.

The processor is an ARM processor embedded on the chip. DDR is off-chip memory. DMA indicates that the data transfer method between memory and CPU and FPGA is direct memory access. The AXI bus is used for data and control signal transfer. The CNN accelerator is a neural network computing part composed of multiple IP cores. To transfer large amounts of data, the off-chip DDR is connected to the CNN accelerator via the AXI4-STREAM bus. The system uses a DMA controller internally to take care of address reading and writing. DMA0 divides the separate read and write data ports, each of which is 64 bits, and is responsible for transferring the feature maps and weights that the CNN accelerator needs to compute, as well as for writing the results of the CNN accelerator's computation to memory. Since DMA1 is only responsible for transferring the results of the previously obtained convolutional layer operations, DMA1 is only responsible for moving data from the main memory to the slave memory, which is located inside the CNN accelerator and is used for the layer-hopping accumulation operations inside the accelerator. The on-chip processor coordinates system operation via the AXI4-Lite bus. The AXI bus between the processor and the CNN accelerator can also be used to transfer parameter types, and since the model in this paper has been quantized in the previous section, only INT types and bool types are transferred.

4.2. CNN Accelerator

4.2.1. Overall Architecture of the CNN Accelerator. The internal design of the CNN accelerator is shown in Figure 6. To enable the full functionality of the network structure in YOLOv3, the IP cores in the CNN accelerator contain: convolutional layer IP, UpSampling layer IP, YOLO layer IP,

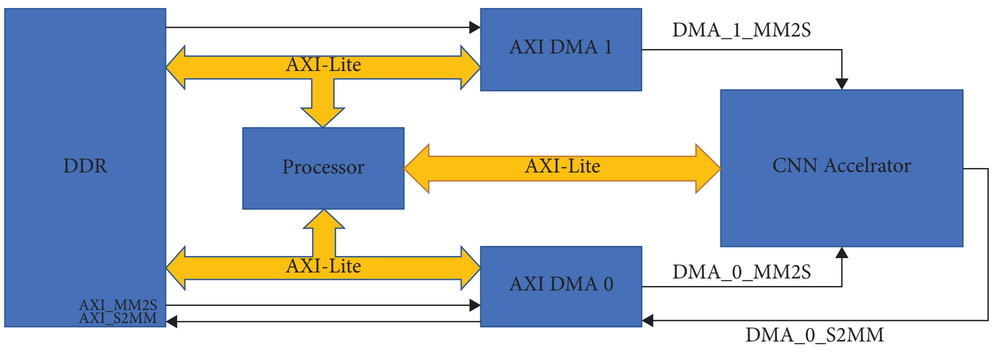


Figure 5: System architecture.

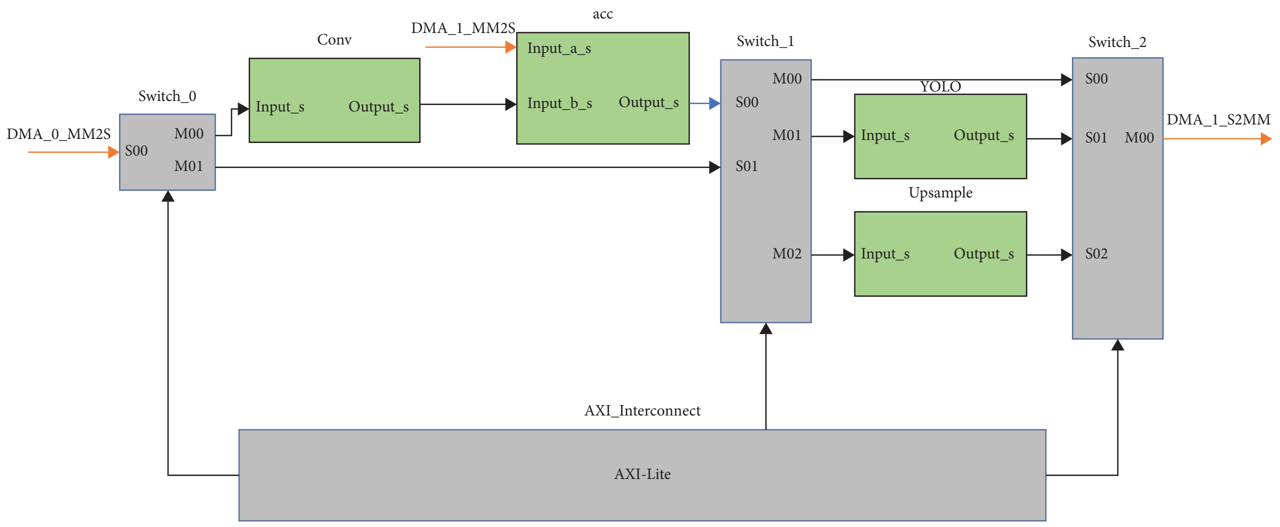


Figure 6: CNN accelerator design.

and acc IP. First, the convolutional layer IP core obtains the weights and input feature maps from DDR by DMA and outputs the output feature maps to the acc IP after convolutional operation. The acc IP implements the calculation function of activation and hopping function. Switches 0 and 1 control whether the DMA0 input flows into the convolutional IP or into other IP cores. Also, switches 1 and 2 determine which layer the output of the convolution operation should be placed on, and these network layers can be YOLO layers, Up-Sampling layers, or direct outputs without any network layers. Finally, the output of the whole process is saved in the main memory by DMA. Among them, since the detection system designed by FPGA only uses the inference part of YOLOv3, this paper chooses to directly merge the BN layer in the convolution operation. Therefore, the IP core of the convolutional layer in this paper is the same as the DBL structure in the YOLOv3 model. Since the minimum structure in YOLOv3 is the DBL structure, such a design can satisfy the whole YOLOv3 deployment on FPGAs.

4.2.2. Convolutional Layer IP Core. Figure 7 shows the internal design of the convolutional layer IP. The interface of the convolutional layer IP is divided into a data port and a control port. The data port is responsible for transmitting the weights, input feature maps, and output feature maps of the data stream. Before calling the convolution kernel IP, the control logic determines whether the input is a feature map or a weight file based on the parameters about the input type transmitted by the C++ program on the PS. After judging, if it is a feature map, it will be stored to buffer 1. If it is a weight file, it will be saved directly into the internal memory in the convolution kernel. If it is a feature map, the corresponding output will be obtained after convolution performed by multiple convolution kernel modules.

Finally, the result is output to the on-chip DDR in the form of AXI-Stream. According to the flow in Figure 5, the loop continues between the off-chip memory and the FPGA until the whole network operation is finished.

Since the convolution kernel in YOLOv3 needs to input multiple rows of data at the same time in one operation, and the convolution kernel slides in a left-to-right, top-to-bottom direction. In this paper, we choose to access the input in one convolution operation as a line buffer. The convolutional kernel modules in Figure 7 are all internally cached with the line buffer as shown in Figure 8. During the top-down sliding of the convolution kernel, two rows of data are reused for each per-row sliding. For each input image, an array of shift registers with the same column and the same row as the size of the convolution kernel is constructed. Whenever the register

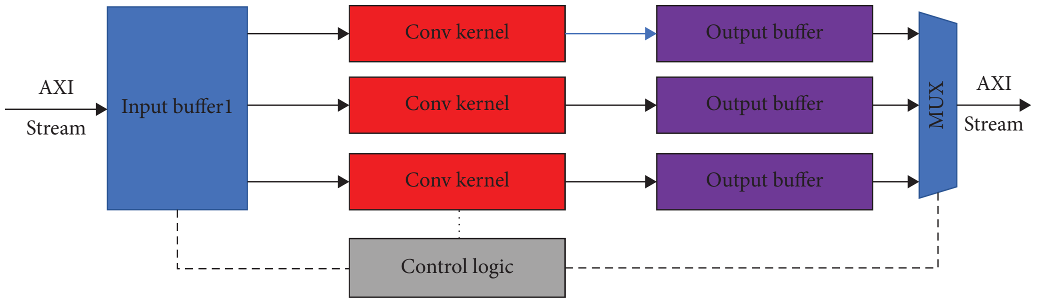


Figure 7: Design of convolutional IP.

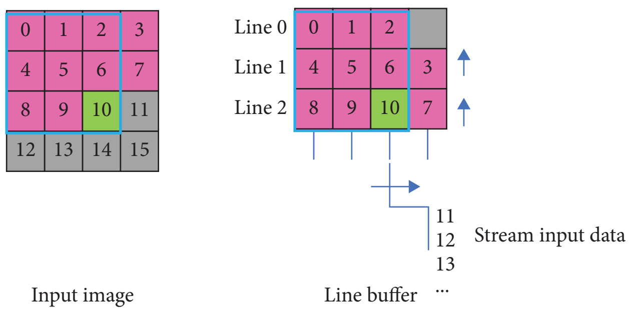


Figure 8: Data allocation of line buffer.

gets an input, the data previously stored in the corresponding column of the input are shifted upward, and then the new number is filled in the vacant position, so that each time the convolution kernel is shifted by row, only the data with the same size of the convolution kernel width need to be filled. Since the size of the convolution kernel in YOLOv3 is only 3×3, and the maximum column size is 418 (the initial input size is 416×416, but the input size will change to 418×418 because of the zero-padding operation during the convolution operation), the maximum row buffer created for each convolution operation is 3×418.

Figure 9 shows the structure inside the convolution kernel module. For a feature map with N number of input channels, N row buffers are created inside the module, and each row buffer corresponds to one channel of the feature map. The sliding window corresponds to the shift window mechanism used in the convolution process to save the 3×3 matrix in the row buffer that needs to be convolved to calculate the 3×3 matrix. C represents the convolutional unit, and the weights and corresponding biases of the convolutional matrix are stored in C. When the sliding window part acquires the matrix to be computed, N convolutional units are computed in parallel. The SUM module is responsible for adding up all the results of the computation. The output results are stored in the output buffer and written to memory at once after the entire feature map is computed.

The internal structure of the convolutional unit C is shown in Figure 10. The convolution unit C expands the input 3×3 feature map matrix and the weight matrix separately. The corresponding multiplications are calculated by nine multipliers in parallel, and the sum of all multiplications is calculated by eight address. The last adder is used to add the bias to the calculated result, and the final value obtained is the result value of one convolution operation.

4.2.3. The acc IP Core. The acc IP core implements the residual part of YOLOv3 and the activation function calculation after the convolution layer. Figure 11 shows the specific design of the acc layer. Since the residual part needs to use the result of the previous convolutional layer operation, it needs to use two input ports, and the output port is still one. The control logic first determines if there are two input streams, if AXI stream b is empty, the accumulation module outputs AXI stream b directly without any operation. If AXI stream b is not empty, it outputs input a and input b after accumulation operation. Since all convolutional layers in YOLOv3 go through the activation function calculation, the output of the accumulation module goes directly to the activation module for the activation function calculation, and the output is still output in the form of AXI-Stream.

4.2.4. Upsampling Layer IP Core. The operation of the upsampling layer is the reverse operation of the pooling layer. For each input data, the upsampling layer in YOLOv3 is expanded into a window of size 2×2, where the upper left corner of the window is the original data and the data in the rest of the window are the same as the original data. The buffers are still line buffer, but for each input, it is first stored inside the line buffer, and then the portion without input of the window is filled. So the maximum line buffer size created by the upper sampling layer IP core is 418×2.

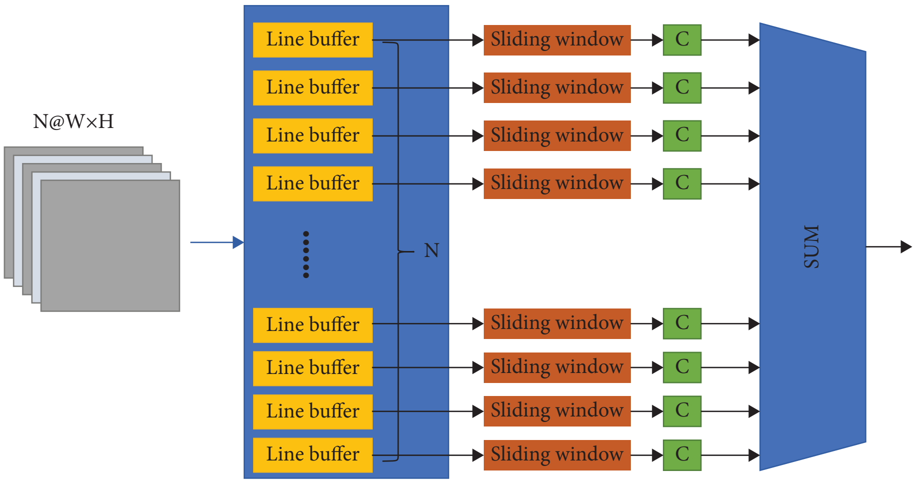


Figure 9: Structure of conv kernel.

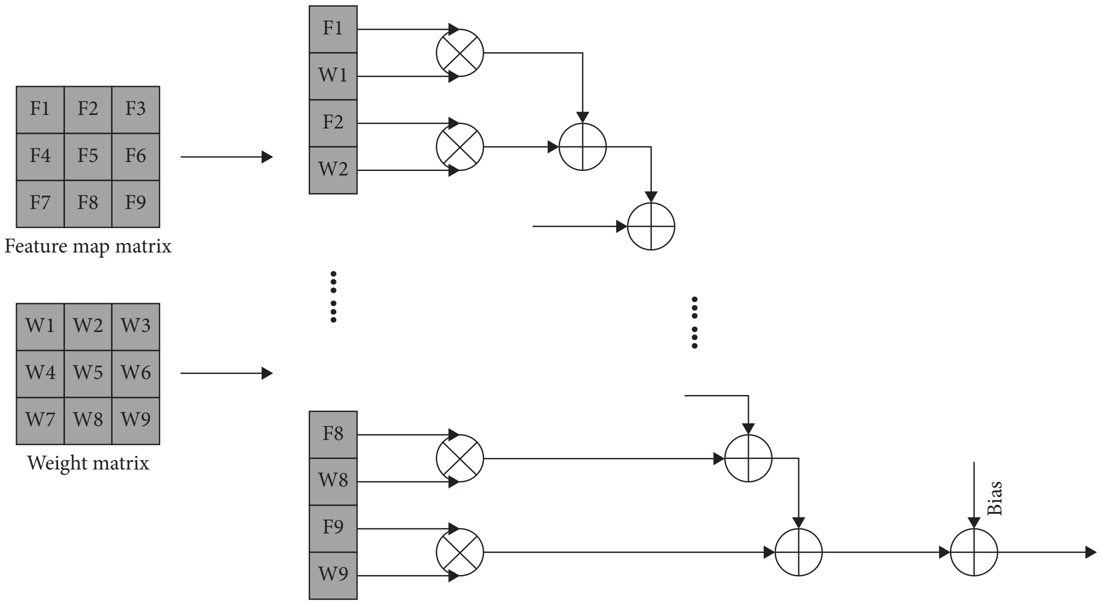


Figure 10: Structure of convolutional calculation unit.

4.2.5. YOLO Layer IP Core. The YOLO layer is used to transform the output feature maps derived from the entire YOLOv3 network into target detection result for different bounding box sizes. The YOLO layer transforms the input into a $W \times H \times C$ size input, that is, the original image is divided into $W \times H$ grids, where the channel C is equal to $(4 + 1 + \text{Class}) \times \text{Numbers}$. Class and Numbers represent the total number of classes predicted by the network and the number of objects predicted by each grid. In this paper, $\text{Class} = 6$, $\text{Numbers} = 3$, $W = H = 13, 26, 52$.

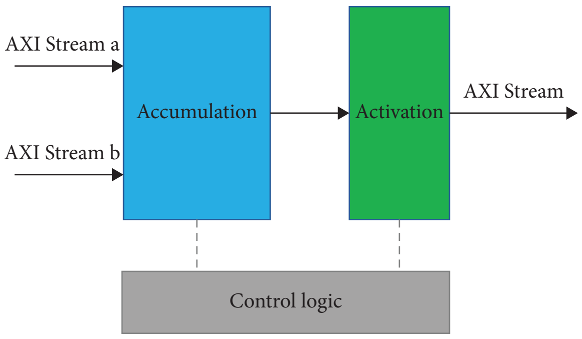


Figure 11: Design of acc IP.

5. Experiments and Results

In this paper, we use ZYNQ7020 to complete the implementation of YOLOv3 acceleration system, and the chip model is 7Z020-2CLG484I. HDMI driver is used to output the detection results on a $1920 * 1080$ resolution screen in real time. For image acquisition, we used XSP-1CA electron microscope to capture the lens video with a frame rate of 60 FPS. The YOLOv3 network was trained and optimized on an NVIDIA GeForce RTX 2080Ti graphics card.

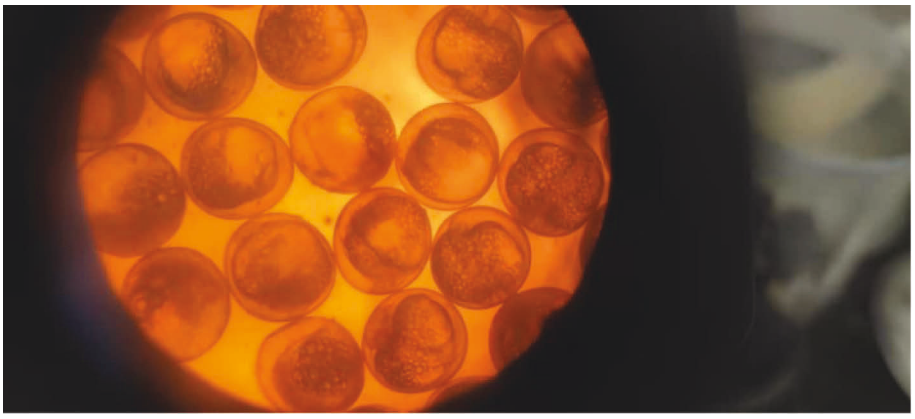


Figure 12: Original image under the microscope.

5.1. Fish Embryo Data set. Since this paper aims at accurate real-time detection of fish embryos, a large number of images of fish embryos are needed to pretrain the network. Therefore, in this paper, images of fish embryos were collected in batches using an electronic microscope in pools 1–10 of phase 2 of the intensive puffer fish farm in Zhongyang, Nantong, Jiangsu Province (120.9°E, 32.6°N). Figure 12 shows a frame from the microscope acquisition video.

As can be seen, the features of the embryo in the figure are more difficult to be extracted directly by the network model and also more difficult to be recognized directly in the system, so it is not advisable to use the YOLOv3 model directly to train the whole image captured by the electron microscope. In this paper, we chose to extract individual embryo images using electron microscope magnification and use LabelImg to annotate each target embryo and make a data set in VOC format. In this paper, we choose to extract individual embryo images after zooming in and use LabelImg to mark each target embryo with annotations (Annotations) to produce a data set in VOC format. A total of 5710 images were collected and divided into training set, test set, and validation set in the ratio of 8 : 1 : 1. Among them, 6 types of embryos are included, respectively, dead, gastrul, neurula, caryolytes, ocular, and olfactory. The individual embryo images in the data set are shown in Table 1. In fact, in the results of embryo detection, the detection accuracy of dead embryos is required to be the highest. In this paper, more than twice as many dead embryos were selected as the training set in the selection of data set for ensuring the accuracy of dead embryos in the detection. Figure 13 shows the embryo images of 6 periods in our data set.

5.2. Embryo Data Set in Different Target Detection Algorithms. As shown in Table 2, in the YOLO family of algorithms, the mAP value of YOLOv3 is smaller than that of YOLOv4 and YOLOv5. But its inference speed is about twice that of YOLOv4 and four times that of YOLOv5. Since the mAP value reflects the degree of overlap between the detection frame and the label, a small loss is acceptable for this application. mAP value and inference speed of Fast-RCNN are both inferior to YOLOv3. MobileNetV3 has the best performance in terms of inference speed, but its accuracy loss is too large. In summary, we choose YOLOv3 as our porting model.

5.3. Basic Training of the YOLOv3 Model. For basic training of the model, we pretrained the model using a 20-class VOC2007 data set in order to enhance the generalization ability of the network model and to speed up the convergence of the model. In basic training and pretraining, the model is saved once for every 100 epochs trained, the learning rate is 0.01, the batch size is 32, the pretraining is 50 epochs, and the basic training is 500 epochs. The pretrained obtained model is trained using the data of embryos. The loss curves of the training process are shown in Figure 14(a). Figure 14(b) shows the loss curves of the training process without pretraining and directly training the model using the embryo data set. Figure 14(b) shows the loss curves of the training process for the model trained directly using the embryo data set after no pretraining.

In Figure 14(a), the model was trained until 500 epochs without convergence. In Figure 14(b), the model has completed convergence by the 300th epoch of training, so pretraining clearly improves the convergence rate of model training. The loss values and mAP values after the model training are completed in both cases and are given in Table 3.

As shown in Table 3, the mAP values and loss values obtained from pretraining are all superior compared to the model without pretraining. Thus, this paper uses VOC2007 data for pretraining before basic training. The trained model achieves 86.72% mAP and 1.922 loss values on the test data set. This indicates that the model has far surpassed the accuracy of traditional manual identification methods for six embryonic periods: dead, gastrul, neurula, caryolytes, ocular, and olfactory. However, the trained model only has an inference speed of 12 fps on the GPU, which is obviously not sufficient for the task of real-time detection and thus requires optimization of the model.

5.4. Model Pruning of the YOLOv3 Model

5.4.1. Channel Pruning. After completing the training of YOLOv3, the optimal model from the base training is sparsely trained according to Equation (1) to facilitate

Table 1: Fish embryo data set.

	Dead	Gastrul	Neurula	Caryolytes	Ocular	Olfactory
Train	1528	896	912	712	814	848
Test	191	112	114	89	101	106
Validation	191	112	114	89	101	106

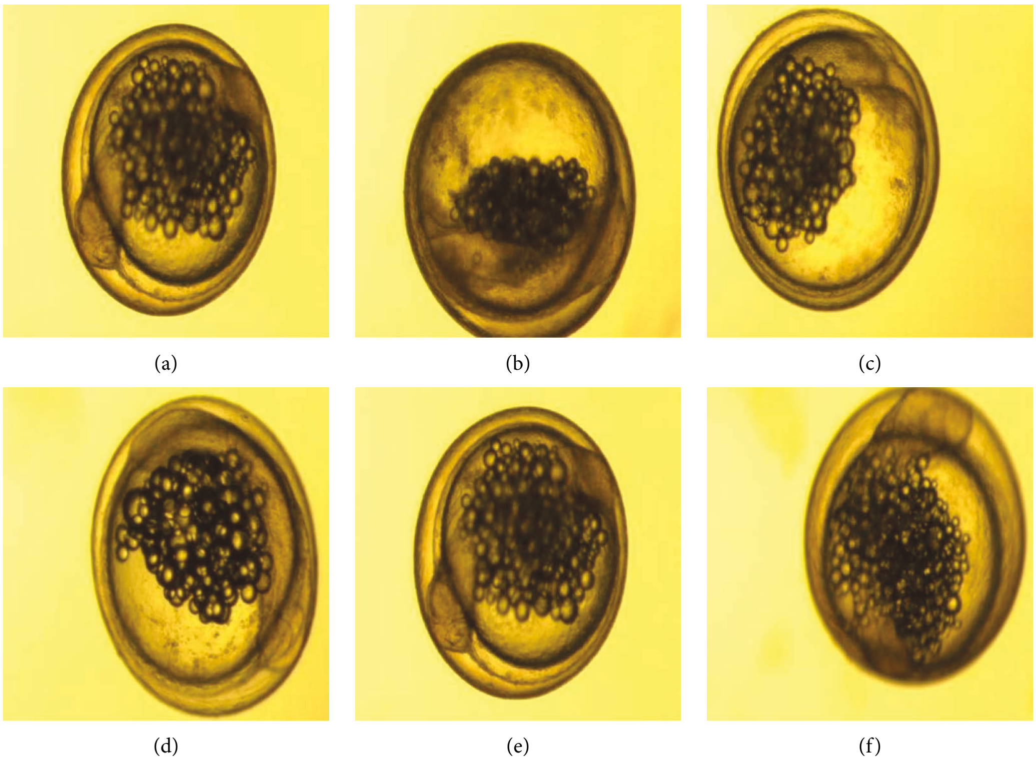


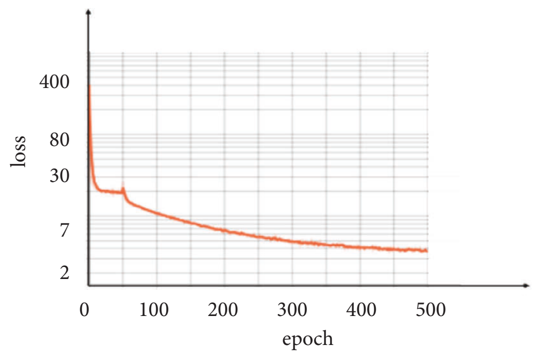
Figure 13: Embryo image: (a) dead embryo, (b) gastrul embryo, (c) neurula embryo, (d) embryo, (e) ocular embryo, and (f) olfactory embryo.

Table 2: Performance of several models trained by the embryo data set.

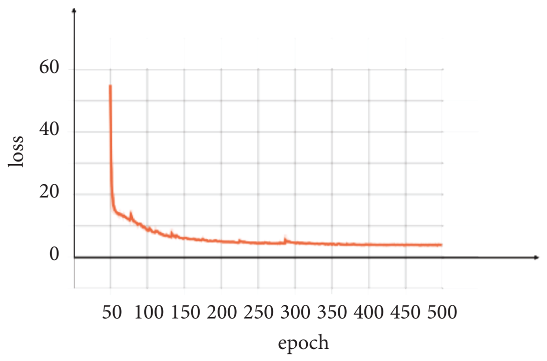
Network	mAP (%)	FPS
YOLOv3	79.93	12.1
YOLOv4	81.22	6.8
YOLOv5	84.51	3.1
Fast-RCNN	76.14	6.4
MobileNetV3	54.33	14.2

channel pruning and layer pruning of the model. The distribution of the corresponding γ parameters in the BN layer during sparse training and basic training is shown in Figure 15, and all four plots in Figure 15 are histograms. The vertical axis in Figures 15(a) and 15(b) plots the number of rounds of model training, the horizontal axis indicates the value of the γ parameter, and the height of the histogram represents the number of γ parameters at this size. As shown in Figure 15(a), in the basic training, as the number of training rounds increases, the distribution of γ parameters still resembles a normal distribution, with the vast majority of parameters in the range of 0.5–1.5. This distribution makes it possible that most of the channels will still remain after channel pruning, and it will make the mean values of γ parameters required for subsequent layer pruning almost all the same, making the process of layer pruning of the model difficult.

Figure 15(b) shows the changes in the distribution of γ parameters during sparse training, in which most of the parameters gradually decrease to close to 0 as the number of training epochs gradually increases, as a way to filter out the channels that can be pruned. Plots (c) and (d) represent the distribution of γ parameters in each DBL structure in the model after basic training and sparse training, respectively. The vertical axis indicates the number of layers in which the specific DBL in its model is located, and the horizontal axis is the size of the γ parameter, and its histogram height indicates the number of γ parameters at this size. After the base training, the parameter distribution of the model is very heterogeneous, and most of the parameter distributions are in the range of 0.5–1.5, making it difficult to perform channel pruning for each layer. In contrast, after sparse training, most of the parameters are close to with 0. This distribution is more convenient for channel pruning. After sparse training, all channels with γ parameters less than 10^{-3} are removed in this paper.



(a)



(b)

Figure 14: Loss curve in training. (a) Loss without pretraining. (b) Loss with pretraining.

5.4.2. Layer Pruning. After channel pruning, the mean values of all the DBL structural γ parameters in the model need to be calculated and layer pruning is performed. Fine-tuning is performed by secondary training after layer pruning. Since the model needs to be deployed to hardware for practical application in this paper, it is necessary to increase the inference speed of the model as much as possible on the basis of ensuring the accuracy of the model. In this paper, we start with 10% pruning rate in increments of 10% and gradually increase the pruning rate. The relationship between mAP value, output video FPS and pruning rate is shown in Figure 16. When the pruning rate of the model exceeds 60%, the mAP value decreases dramatically, and this change cannot be corrected by retraining. Because the model eventually needs to be deployed to the hardware, a balance point of inference speed and accuracy is chosen, and the pruning rate chosen in this paper is 50%. That is, the network layers with the last 50% of the γ parameter size ranking are deleted. It is equivalent to subtracting 26 DBLs (i.e., subtracting 26 convolutional layers) to reduce the model by about half of the parameters and computational effort.

Since there was a drop in model accuracy after pruning, this paper chooses to fine-tune the model using the original data set as the fine-tuning data set. After retraining 50 epochs with the original training set at 1%, the mAP value of the model on the test set rebounded to 86.14%, where the model size was reduced to 110.4 MB and the video inference speed on GPU could reach 17.4 FPS.

5.5. Model Quantization of the YOLOv3 Model. After the model pruning, the 32 bit floating-point parameters of the model were chosen to be quantized into INT4 and INT8 models all through Algorithm 1. The quantization results are shown in Table 4.

The data in Table 4 are the individual performance metrics of the model quantized by Algorithm 1 for INT8 quantization and INT4 quantization, respectively, the mAP value, the model size, and the FPS at inference. The data in Table 4 are the individual performance metrics of the model quantized by Algorithm 1 for INT8 quantization and INT4 quantization, respectively, the mAP value, the model size, and the FPS at inference. It can be seen that the quantization method of Algorithm 1 can significantly improve the inference speed of the model while ensuring little change in model accuracy. The INT8 quantized model achieves 83.52% mAP and 33.9 fps on the GPU. The INT4 quantized model achieves 82.49% mAP on the GPU and 50.3 fps on the GPU, which is a 32.9 fps improvement over the pruned model. After quantization by INT4 and INT8 of Algorithm 1, the two quantization methods only have a difference of about 1.03% in mAP. In fact, in the actual detection process, about 1% of mAP may make 1 out of 1000 embryos misclassified, which has a negligible impact on the detection accuracy of the actual system application, but the network model after INT4 quantization is nearly 17 fps higher in GPU inference speed of processing. In terms of model size, INT4 quantization can reduce the model size to 1/8 of the original size, making it easier to store the model. Since this paper has the requirement of real-time detection for the system deploying YOLOv3 on FPGA, all the YOLOv3 models are quantized into 4 bit fixed points.

5.6. FPGA Implementation of Optimized YOLOv3. After finishing the optimization of the whole YOLOv3 model, the weight parameters of the optimized model are stored into the DDR at the PS side of ZYNQ in advance. According to the content in Section 3.2, we package the IP core in Vivado and connect it using AXI bus protocol to complete the hardware implementation of the YOLOv3 model. Figure 17 shows the specific design of the YOLOv3 gas pedal in Vivado corresponding to Figure 6. The hardware acceleration is accomplished by splitting each YOLOv3 network layer into its corresponding network structure and passing in weight parameters at the time of use. In which, as designed in Chapter 3, the convolutional layer passes through switch 1 and selectively enters the subsequent yolo layer or upsample layer, where the acc IP core after the conv IP core is used to handle the accumulation operation of the results output from the two different network layers shown in Figure 7 and to implement the activation function LeakyRelu.

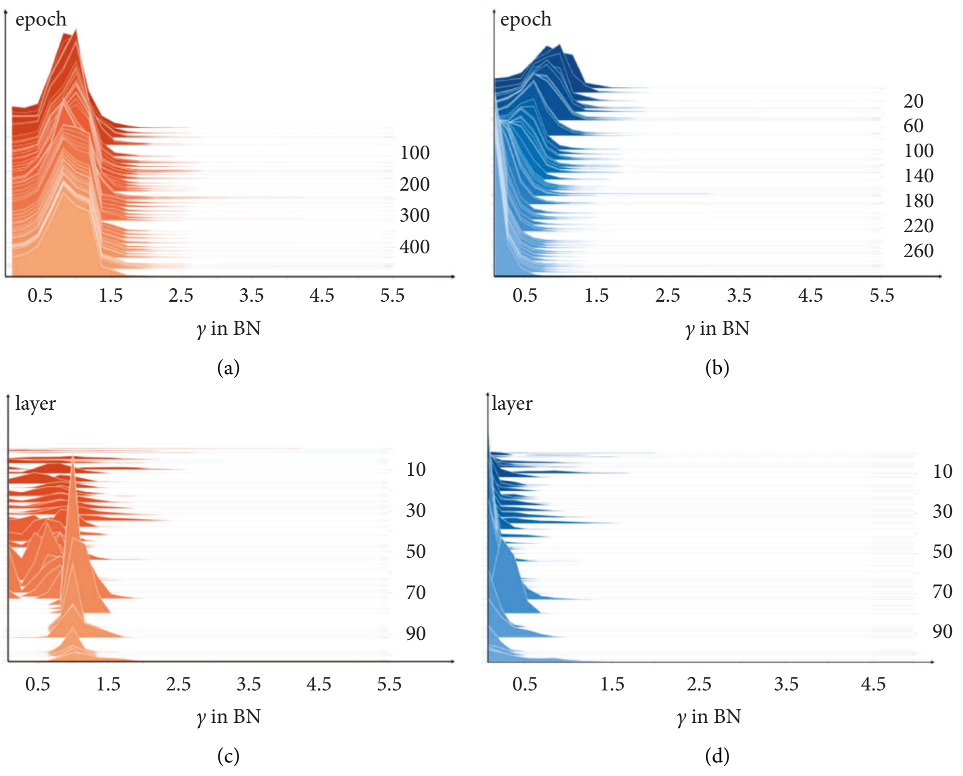


Figure 15: γ parameter distribution. (a) Distribution of γ parameters during basic training, (b) distribution of γ parameters during sparse training, (c) distribution of all γ parameters of the whole model after basic training is completed, and (d) distribution of all γ parameters of the whole model after sparse training is completed.

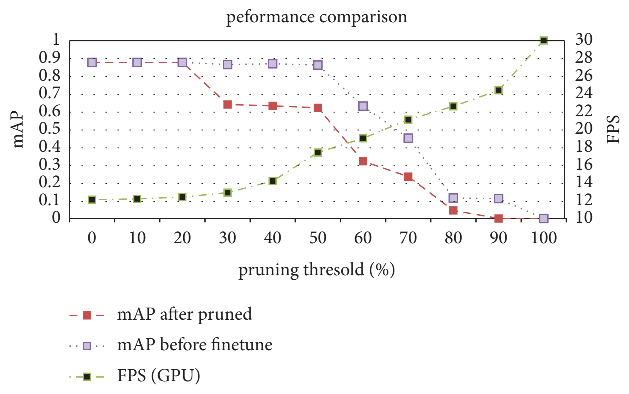


Figure 16: mAP and FPS in different model pruning thresholds.

Table 3: mAP and loss comparison.

	Loss	mAP (%)
Pretrained model	1.922	86.72
Without pretrained model	3.4368	79.93

Table 5 shows the resource utilization of the YOLOv3 gas pedal system built in this paper on the FPGA platform. In terms of resource utilization, the resource utilization of LUT, BRAM, DSP, and FF in this paper is 51.78%, 82.27%, 66.07%, and 47.75%, respectively. The LUT, DSP, and FF parts are used to process numerical and logical operations. The BRAM is used to store feature maps and weight inputs. As can be seen, the YOLOv3 accelerator in this paper is built using almost all the resources on the chip in order to achieve the maximum inference speed. Such a hardware architecture can complete the inference process of YOLOv3 more quickly.

The performance of the detection model on different types of embedded devices is given in Table 6, where we use CUDA10.1 with Tensorflow2.4 to accelerate the inference speed of the YOLOv3 optimized model on RTX-2080Ti and ARM, and the ARM architecture uses the Cortex-A9 dual-core ARM currently used by most

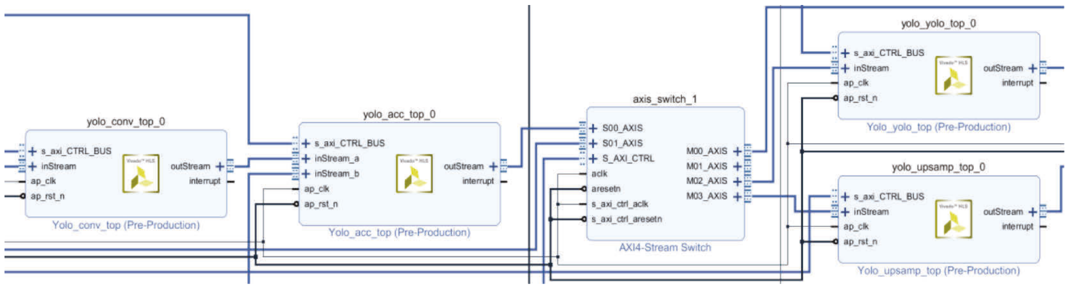


Figure 17: YOLOv3 accelerator implementation in Vivado.

Table 4: Indicators of the quantization model.

	mAP (%)	Model size (MB)	FPS on GPU
Pruned model	86.14	110.4	17.4
INT8 quantization	83.52	24.9	33.9
INT4 quantization	82.49	12.8	50.3

Table 5: Performance of the embryo detection model on different types of embedded devices.

	LUT	BRAM	DSP	FF
Used	27548	50801	181	50801
Available	53200	106400	220	106400
Utilization	51.78%	66.07%	82.27%	47.75%

Table 6: Performance in different embedded devices.

Device	Power (w)	FPS	Processing time per image (ms)
RTX-2080Ti	230	50.3	7.9
ARM Cortex-A9 dual-core	1.6	2.7	246
ZYNQ-7020	2.524	34.1	12.1

embedded systems. FPGA hardware acceleration is used to improve the model inference speed. Since the optimized model is used on all devices in this paper for comparison on the same test set, there will be no gap in the mAP value reflecting the accuracy of the model itself. The FPGA accelerated system built on ZYNQ in this paper can reach about 5 times the inference speed of pure CPU inference, although the power consumption is slightly higher than that of the inference system with ARM for the model. With 24 frames as the real-time detection standard, this system steadily achieves the effect of real-time inference. Compared with GPU, this paper lacks about 50% of the image inference speed. However, in terms of application, the power consumption of GPU reaches nearly 92 times of the power consumption in this paper compared to this paper, and it is impossible to meet this power consumption demand in actual aquaculture deployment. For embedded deployment, the system in this paper not only saves cost compared to traditional GPU + CPU and pure CPU for aquaculture but also satisfies basically all fish embryo detection needs at the same time.

5.7. Result. For each input embryo image acquired from the microscope, we scaled it to the input size of 416×416 required by the optimized YOLOv3 model. After input to the optimized YOLOv3 model, based on the obtained 3 feature map results, the nonmaximal value suppression algorithm was used on the PS side to eliminate the duplicate detection results and get the most accurate results. Some sample embryo detection images are shown in Figure 18.

The corresponding FPGA on-chip real-time detection system effect and system structure are shown in Figure 19, the whole real-time detection system consists of three parts, namely ZYNQ chip responsible for CPU control and YOLOv3 acceleration, electronic microscope responsible for embryo data acquisition, and HDMI display responsible for image output.

The experimental results show that the system is able to acquire the results of embryo detection correctly and in real time. The inference speed of 34.1 fps on average is achieved with 2.524w of power consumption at 100 MHz FPGA operating frequency, while 82.49% of mAP value can be achieved in detection accuracy. Compared with the

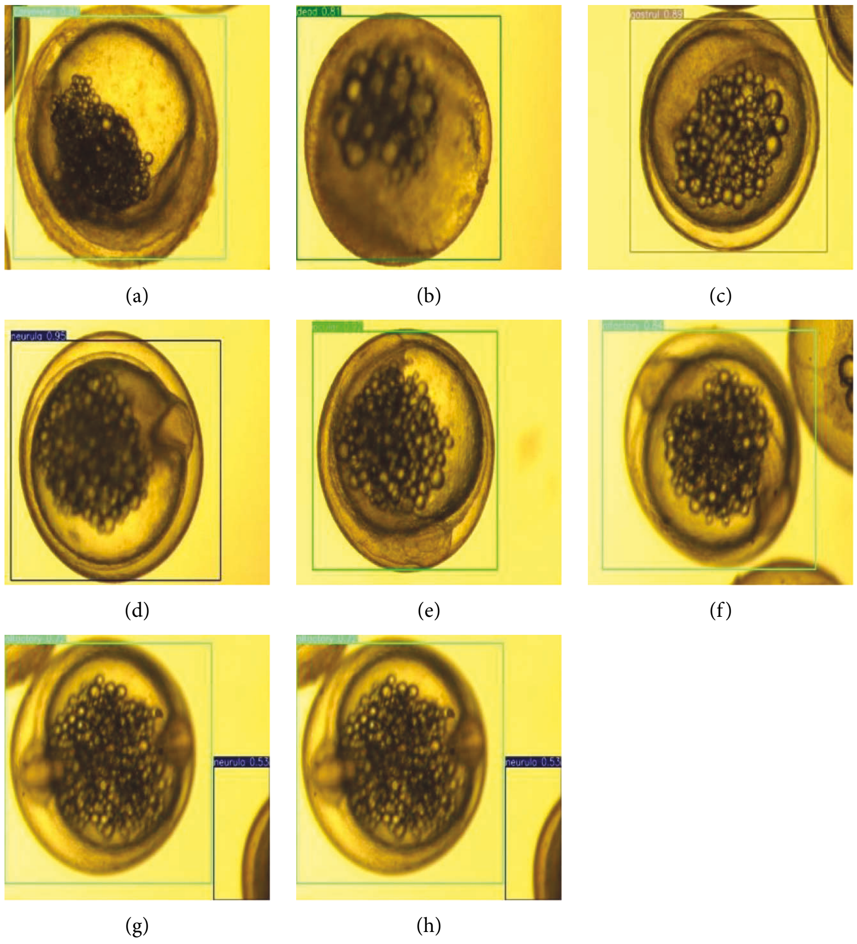


Figure 18: Embryo detection by the optimized model. (a) Caryolites, (b) dead, (c) gastrul, (d) neurula, (e) ocular, and (f–h) olfactory.

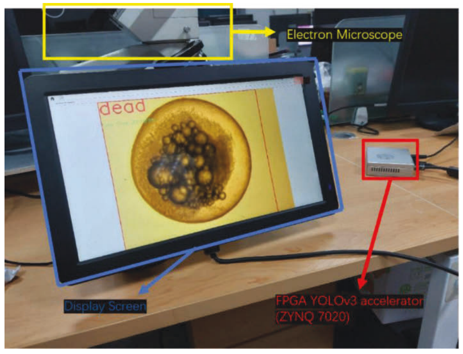


Figure 19: Real-time fish embryo detection system.

original model deployed directly with CPU + GPU, the mAP value is only 4.23% lower, but the real-time detection can be completed with nearly 1/90th of the power consumption.

6. Conclusion

In the paper, an FPGA-based system for fish embryo detection is proposed. Based on using YOLOv3 as the basic model, the model itself is optimized by combining a model pruning method based on scaling factors and a quantization method with trainable thresholds, and the optimized model is deployed on an FPGA platform to realize a real-time fish embryo detection system. Compared with the direct use of the original YOLOv3 model, a better compromise is achieved in terms of inference speed and inference accuracy, and the on-chip computing resources are utilized to the maximum. This FPGA-based fish embryo detection system fully satisfies the current demand for embedded devices in fish farming and facilitates the shift from the use of high-volume experienced staff to artificially intelligent embedded devices for embryo detection and rapid analysis of embryos in farming. The system has been applied in the embryo farming of Zhongyang Pufferfish Estate in Nantong, Jiangsu Province. It has achieved 34.1 fps with 82.49% mAP value for real-time accurate embryo detection at 2.524w power consumption.

Data Availability

All data used in this study are included in the manuscript.

Conflicts of Interest

The authors declare no conflicts of interest.

Acknowledgments

This research was funded by Guangdong Key Areas R&D Program Projects, under grant no. 2021B0202070001.

References

[1] J. Chen, S. Fangfang, J. Zhang, and F. Yongjie, "Zhao Daoquan," *Jiangsu Agricultural Science*, vol. 49, p. 45, 2021.

[2] M. Ashaf-Ud-Doulah, S. M. M. Islam, M. M. Zahangir, M. S. Islam, C. Brown, and M. Shahjahan, "Increased water temperature interrupts embryonic and larval development of Indian major carp rohu Labeo rohita," *Aquaculture International*, vol. 29, no. 2, pp. 711–722, 2021.

[3] L. Ma, R. Dessiatoun, J. Shi, and W. R. Jeffery, "Incremental temperature changes for maximal breeding and spawning in Astyanax mexicanus" Jove-Journal of Visualized Experiments," 2021.

[4] S. E. Cordova-de la Cruz, G. Martinez-Bautista, E. S. Pena-Marin et al., "Morphological and cardiac alterations after crude oil exposure in the early-life stages of the tropical gar (Atractosteus tropicus)," *Environmental Science and Pollution Research*, vol. 29, no. 15, pp. 22281–22292, 2021.

[5] R. L. Naylor, R. W. Hardy, A. H. Buschmann et al., "A 20-year retrospective review of global aquaculture," *Nature*, vol. 591, no. 7851, pp. 551–563, 2021.

[6] J. Clarke, "Live imaging of development in fish embryos," *Seminars in Cell & Developmental Biology*, vol. 20, no. 8, pp. 942–946, 2009.

[7] O. Ishaq, S. K. Sadanandan, and C. Wahlby, "Deep fish," *SLAS Discovery*, vol. 22, no. 1, pp. 102–107, 2017.

[8] H. T. Rauf, M. I. U. Lali, S. Zahoor, S. Z. H. Shah, A. U. Rehman, and S. A. C. Bukhari, "Visual features based automated identification of fish species using deep convolutional neural networks," *Computers and Electronics in Agriculture*, vol. 167, p. 105075, 2019.

[9] A. M. Naderi, H. Bu, J. Su et al., "Deep learning-based framework for cardiac function assessment in embryonic zebrafish from heart beating videos," *Computers in Biology and Medicine*, vol. 135, Article ID 104565, 2021.

[10] P. Dey, "The emerging role of deep learning in cytology," *Cytopathology*, vol. 32, no. 2, pp. 154–160, 2021.

[11] B. H. Tang, Z. X. Pan, K. Yin, and A. Khateeb, "Recent advances of deep learning in bioinformatics and computational biology," *Frontiers in Genetics*, vol. 10, p. 214, 2019.

[12] Y. Li, Z. G. Luo, N. Y. Guan et al., *Progress in Biochemistry and Biophysics*, vol. 43, p. 472, 2016.

[13] C. Qian, M. Tong, X. Yu, and S. Zhuang, "CNN-based visual processing approach for biological sample microinjection systems," *Neurocomputing*, vol. 459, pp. 70–80, 2021.

[14] Q. Xu, Z. Wang, F. Wang, and Y. Gong, "Multi-feature fusion CNNs for Drosophila embryo of interest detection," *Physica A: Statistical Mechanics and Its Applications*, vol. 531, p. 121808, 2019.

[15] D. Dirvanauskas, R. Maskeliunas, V. Raudonis, and R. Damasevicius, "Embryo development stage prediction algorithm for automated time lapse incubators," *Computer Methods and Programs in Biomedicine*, vol. 177, pp. 161–174, 2019.

[16] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: survey and challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, pp. 135–253, 2007.

[17] M. Xia, Z. Huang, L. Tian et al., "SparkNoC: an energy-efficiency FPGA-based accelerator using optimized lightweight CNN for edge computing," *Journal of Systems Architecture*, vol. 115, Article ID 101991, 2021.

[18] N. Y. Khanday and S. A. Sofi, "Taxonomy, state-of-the-art, challenges and applications of visual understanding: a review," *Computer Science Review*, vol. 40, Article ID 100374, 2021.

[19] A. G. Blaiech, K. Ben Khalifa, C. Valderrama, M. A. C. Fernandes, and M. H. Bedoui, "A survey and taxonomy of FPGA-based deep learning accelerators," *Journal of Systems Architecture*, vol. 98, pp. 331–345, 2019.

[20] T. Li, T. Zhang, G. Yu, J. Song, and J. Fan, "Minimizing temperature and energy of real-time applications with precedence constraints on heterogeneous MPSoC systems," *Journal of Systems Architecture*, vol. 98, pp. 79–91, 2019.

[21] V. R. S. Mani, A. Saravanaselvan, and N. Arumugam, "Performance comparison of CNN, QNN and BNN deep neural networks for real-time object detection using ZYNQ FPGA node," *Microelectronics Journal*, vol. 119, Article ID 105319, 2022.

[22] V. Sharma and R. N. Mir, "A comprehensive and systematic look up into deep learning based object detection techniques: a review," *Computer Science Review*, vol. 38, Article ID 100301, 2020.

[23] I. Damaj, S. K. Al Khatib, T. Naous, W. Lawand, Z. Z. Abdelrazzak, and H. T. Mouftah, "Journal of King Saud University - Computer and Information Sciences," *Open access journal*, vol. 12, 2021.

[24] M. Zhao, C. Hu, F. Wei, K. Wang, C. Wang, and Y. Jiang, "Real-time underwater image recognition with FPGA embedded system for convolutional neural network," *Sensors*, vol. 19, no. 2, p. 350, 2019.

[25] X. Wei, W. Liu, L. Chen, L. Ma, H. Chen, and Y. Zhuang, "FPGA-based hybrid-type implementation of quantized neural networks for remote sensing applications," *Sensors*, vol. 19, no. 4, p. 924, 2019.

[26] Y. Luo and Y. Chen, "FPGA-based acceleration on additive manufacturing defects inspection," *Sensors*, vol. 21, no. 6, p. 2123, 2021.

[27] M. R. Al Koutayni, V. Rybalkin, J. Malik et al., "Real-time energy efficient hand pose estimation: a case study," *Sensors*, vol. 20, no. 10, p. 2828, 2020.

[28] A. F. J. Redmon, "YOLOv3: An Incremental Improvement", https://arxiv.org/pdf/1804.02767.pdf, 2018.

[29] K. H. S. Ren, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks" https://arxiv.org/abs/1506.01497, 2015.

[30] A. G. H. Z. C. K. W. W. A. Adam, 2017 "MobileNets: efficient convolutional neural networks for mobile vision applications" https://arxiv.org/abs/1704.04861.

[31] E. A. Smirnov, D. M. Timoshenko, and S. N. Andrianov, "Comparison of regularization methods for ImageNet classification with deep convolutional neural networks," *AASRI Procedia*, vol. 6, pp. 89–94, 2014.

[32] C. Wu, V. Fresse, B. Suffran, and H. Konik, "Accelerating DNNs from local to virtualized FPGA in the Cloud: a survey of trends," *Journal of Systems Architecture*, vol. 119, p. 102257, 2021.

[33] A. G. Sambhav R Jain, M. Wu, and C. Dick, 2019, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks" https://arxiv.org/abs/1903.08066.