Tail Layer CNN Training for a SoC-based FPGA

Yuki Takashima
Tokyo Institute of Technology, Japan
takashima.y.ad@m.titech.ac.jp

Akira Jinguji
Tokyo Institute of Technology, Japan

Ryosuke Kuramochi
Tokyo Institute of Technology, Japan

Ryota Kayanoma
Tokyo Institute of Technology, Japan

Hiroki Nakahara
Tokyo Institute of Technology, Japan

Abstract— The demand for deep learning has increased, and many accelerators have been proposed. Although they perform inference at high speed, many of them have problems in training. We present the ”tail layer training” for a convolutional neural network (CNN). It is implemented with DPU, a CNN accelerator provided by Xilinx, and a CPU. It retains a high computational speed compared to conventional methods because most of the CNN can be processed by the accelerator. On the other hand, only the tail layer is updated by the CPU, enabling the weights to be updated or added. Since the number of neurons and classes in the output must be the same for image classification, it is effective for retraining to count the number of classes. We found that domain similarity between existing classes and classes to be added is important in CIFAR10 and ImageNet datasets for tail layer training. Accuracy loss is negligible when training only the tail layer with two added categories. The processing speed reduction was almost negligible. Our scheme can be applied to the all existing SoC-FPGA-based CNN accelerator.

I. Introduction

Deep learning models are widely used in a variety of fields, with image classification at the top of the list, for example Classification[1], Object Detection[2], Face Detection[3], Segmentation[4], Pose Estimation[5], Molecular Depth Estimation[6], Super Resolution[7], GAN: Generative Advisal Network[8], etc. Most of these applications are based on Convolutional Neural Network (CNN)[9].

CNNs need a massive number of parameters and computational complexity compared to existing machine learning models. Therefore, while they are suitable for training complex tasks with large amounts of data, they are also computationally demanding. Thus, CNN-specific accelerators have been developed and used in embedded systems that require high speed and low power consumption. However, many accelerators cannot train on the accelerator because they require pre-optimization (quantization and sparsification) and compilation of the trained model. In FPGAs, DPU (Deep learning Processor Unit) developed by Xilinx can be used for fast inference with learned models, but it does not support training. In contrast, embedded systems may require training on edge after design with additional functionality or updates. For example, face recognition has recently been developed as a seamless authentication method. Face recognition systems are expected to be used in various locations, so there may be some situations where the system is not connected to a network. In this case, it is difficult to train a new face for authentication using only an edge terminal.

This paper proposes a method to implement additional hardware and training algorithms in a DPU, one of the CNN accelerators, and perform other training on the CPU. Typically, CNNs are trained using backpropagation. It calculates the error between the training data and the correct data for all layers and updates the parameters while controlling overfitting by the learning rate. Therefore, it requires many computer resources and time and is unsuitable for embedded systems. In contrast, adding or updating functionality in embedded systems is often limited, and the tasks are often the same with limited additional data. Fine-tuning is a method for fast convergence of CNN training. It replicates some of the parameters of the trained model to a part of the CNN model to be trained. Thus, making the trend of the target model closer to that of the trained model. For example, fine-tuning on ImageNet can converge in a short training epoch in image recognition tasks. In this paper, we target a classification task and propose to add a training function to the accelerator by making only the tail layer of the CNN independent. Only the tail layer is trained in software using an ARM processor on an FPGA, and the trained parameters are reflected in the hardware dedicated to the tail layer. In the classification task, the first part of the CNN is considered the feature extraction of the image. The last part is considered to be the encoding of the extracted features into a class index that humans can understand. If this assumption is correct, the classification task can be re-trained by implementing a pre-trained model in the same domain with a DPU and training only the encoding part at the tail layer.

The structure of this paper is as follows. Chapter 2 describes related research and tools such as CNN, PCA. Chapter 3 describes the proposed method of tail layer training. Chapter 4 presents experiments and a discussion of the proposed method, and Chapter 5 concludes the paper.

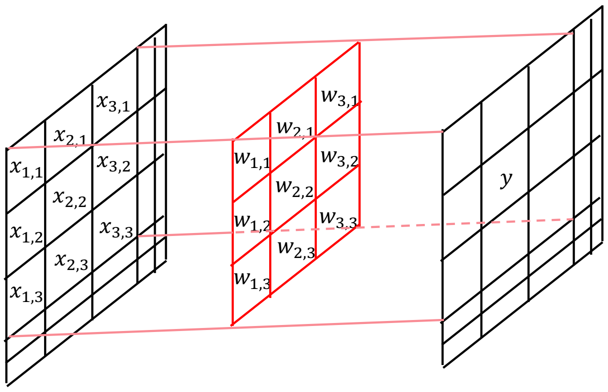

Fig. 1. Convolutional Layer Overview.

II. Related Work

A. CNN (Convolutional Neural Network)

A CNN (Convolutional Neural Network) is a deep learning model used for tasks such as image classification, which automatically extracts features from images by performing two-dimensional convolution. Typically, models used for image classification require that the number of neurons in the output layer matches the number of classification classes. Therefore, the CNN must be re-trained with all the data when adding a class.

CNN models such as VGG[10], ResNet[11], and MobileNetV2[12] have already been proposed, which use a convolutional layer, a pooling layer, and a fully connected layer. The convolutional layer (conv layer) has a structure like Fig. 1. The output is determined by the sum of the product of the weights assigned to the convolution window and the input, and this process is applied by shifting the window. It has the effect of extracting a variety of features. The pooling layer has the effect of compressing the size of the input by obtaining representative values. There are two methods for selecting expected values: max pooling (selecting the maximum value within a specific range) and average pooling (setting the average value). These are weightless because they are determined from the input values only. In the fully connected layer (fc layer), the output is determined from the sum of the product of the inputs and weights. Still, there are as many weights as the product of the number of input dimensions and the number of output dimensions. In VGG, ResNet and MobileNetV2, the all-connected layer is used at the end of the model.

B. PCA (Principal Component Analysis)

Principal Component Analysis (PCA) is a method of changing the dimensions of multi-dimensional data while preserving as many of its characteristics as possible. The axis that best represents the information content of the original data is the first principal component, the axis orthogonal to the first principal component and meaning the second most information content is the second principal component, and so on. Since the dimension can be changed arbitrarily while maintaining the characteristics as much as possible, the data can be compressed to two dimensions and plotted on a planar diagram to confirm the relationship between the data visually.

III. Proposed Method

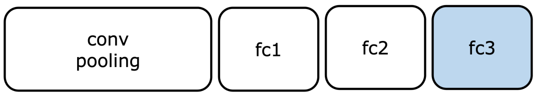

Fig. 2. Tail Layer (fc3) Training in VGG16.

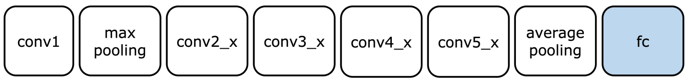

Fig. 3. Tail Layer (fc) Training in ResNet50.

A. Tail Layer Training

The number of neurons in the output layer of the CNN must match the number of classes to be classified. Therefore, when additional classes are desired, the model must be changed to one with an increased number of output neurons and retrained.

Assuming that features can be extracted while the input passes through many layers of the CNN, it is possible to achieve sufficient accuracy by simply retraining the mapping between features and labels in the tail layer. This training is defined as tail layer training. In this paper, we propose two methods for training the tail layer: one is to use a randomly determined initial value, and the other is to train only the additional classes without updating the weights of the existing classes.

Fig. 2 shows an example of tail layer training when VGG16 is used. Fig. 3 shows an example of tail layer training when ResNet50 is used. The layers shown in white in Fig. 2 and Fig. 3 are the layers whose weights are not updated in the tail layer training. Therefore, only inference is performed on the input images. In contrast, the tail layer shown in blue is the fully connected layer (fc layer) for both VGG16 and ResNet50.

The inference-only layer reuses the same weights. Therefore, using pre-trained weights and compiling only the portions of VGG and ResNet without the tail layer can be computed faster using DPUs. Since this part of the model represents a large portion of the model, it can significantly reduce the computational cost compared to training the entire model. Since this part of the model represents a large portion, it can dramatically reduce the computational cost compared to training the whole model. Since this part of the model represents a significant portion. For example, VGG16 has 134M parameters. Of these, the input for the tail layer has 4,096 dimensions. Therefore, the number of parameters in the tail layer $param_{\mathrm{fin}}$ can be said to be

$$\begin{aligned} param_{\mathrm{fin}} &= 4096 \times 1000 + 1000 \qquad (1)\\ &\simeq 0.41\mathrm{M}. \qquad (2) \end{aligned}$$

Only about 0.3% of the total parameters need to be trained. In addition, the output of the layers whose weights do not change as the training progresses. The training process can be made even faster in the second and subsequent epochs by saving the output at the DPU when training with the same image.

B. *Tail Layer Training to Use Randomly Determined Initial Value*

This method does not use the weights used in the original model but gives the randomly determined initial value. The tail layer is discarded, and a new layer is attached with a different number of output neurons. Instead of discarding parameters that are already accurate, the advantage is that weights can be trained, including the balance between additional classes and existing classes. This method is sometimes called transition learning.

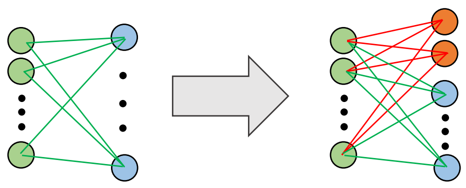

Fig. 4. Example of Training Only the Additional Classes without Updating the Weights of the Existing Classes.

C. *Tail Layer Training with Training Only the Additional Classes without Updating the Weights of the Existing Classes*

We also propose a method for training only the additional classes without updating the weights for the existing classes. In other words, since the weights are already sufficiently accurate, they are reused.

A schematic diagram of this training method is shown in Fig. 4. Here, the left side of the Fig. 4 shows the weights before retraining for the tail layer, the fully connected layer, and the right side shows the weights after retraining. The green circle represents the input of the tail layer, the blue circle represents the neuron corresponding to the class that exists from the beginning among the outputs of the tail layer, and the red circle represents the neuron in the tail layer corresponding to the class be added. The line segments connecting the neurons represent weights, and the green weights corresponding to existing classes remain unchanged before and after training.

By learning only the additional classes, only the red weight updates need to be computed instead of the green weight updates in Fig. 4. It can significantly reduce the computational cost when optimally implemented in hardware. On the other hand, in the image classification task, the output of the entire model is determined by the softmax of the output of the tail layer. Therefore, depending on the balance with the not updated weights, the accuracy may be worse than when training from initial values given by random numbers.

D. *Back Propagation for Tail Layer Training*

Neural networks, including CNNs, cannot make appropriate inferences about the input unless the weights have appropriate values, and the expected output and the model output will not match. Therefore, it is necessary to update the weights, one of which is the back propagation method. Let $\boldsymbol{y}$ be the expected output and $\hat{\boldsymbol{y}}$ be the model output, and let E be the error function $E = |\boldsymbol{y} - \hat{\boldsymbol{y}}|^2$.

The weights are changed to minimize E because E should be adjusted to be zero and $E \geq 0$ is always true in order to make the model output the expected output.To minimize E, the difference Δw for modifying each weight w is updated according to

$$\Delta w = -\eta \frac{\partial E}{\partial w} \qquad (3)$$

using an appropriate learning rate η.

Let z_i^{l-1} be the input of the i-th neuron in the $l-1$ layer and x_i^{l-1} be its output. Also, let $w_{i,j}^{l-1}$ be the weight from the i-th neuron in layer $l-1$ to the j-th neuron in layer l. When the activation function σ is

$$x_j^l = \sigma(\sum_i z_i^{l-1} w_{i,j}^{l-1}) \qquad (4)$$

, then the right side of Eq.(3) is

$$-\eta \frac{\partial E}{\partial w_{i,j}^l} = -\eta \frac{\partial E}{\partial x_j^{l+1}} \sigma'(z_j^{l+1}) x_j^l \qquad (5)$$

.

In the tail layer, for $\frac{\partial E}{\partial x_j^{l+1}}$ in Eq.(5), when the number of layers of the CNN is n, $x_j^n = \hat{y}_j$. By $E = |\boldsymbol{y} - \hat{\boldsymbol{y}}|^2$, it can calculated as

$$\begin{aligned}
\Delta w_{i,j}^{l} &= -\eta \frac{\partial E}{\partial w_{i,j}^{l}} && (6)\\
&= -\eta \frac{\partial E}{\partial x_{j}^{l+1}} \sigma'(z_{j}^{l+1}) x_{j}^{l} && (7)\\
&= -\eta \frac{\partial |\boldsymbol{y} - \hat{\boldsymbol{y}}|^{2}}{\partial \hat{y}_{j}} \sigma'(z_{j}^{l+1}) x_{j}^{l} && (8)\\
&= 2\eta (y_{j} - \hat{y}_{j}) \sigma'(z_{j}^{l+1}) x_{j}^{l} && (9)
\end{aligned}$$

. This shows that only the tail layer can be trained.

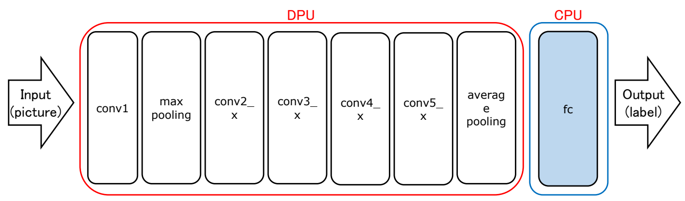

Fig. 5. Hardware Configuration of Tail Layer Training.

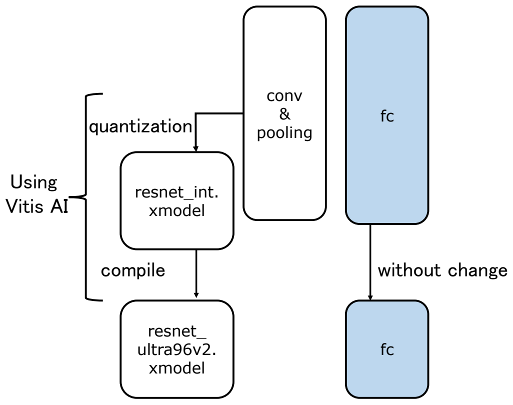

Fig. 6. Model Creation Flow.

E. *Implemented Platforms*

The hardware configuration for tail-layer training is shown in Fig. 5, where models compiled from ResNet except for the tail layer are computed on a DPU. On the other hand, the tail layer for training is computed on the CPU.

The model creation method is shown in Fig. 6. Except for the tail layer of ResNet, it needs to be converted into a format executable by DPU. xmodel files are generated by compiling after quantization using Vitis AI and then transferred to the board to be read. The system can be used for the following purposes. The tail layer can be loaded with weights trained in advance.

IV. Experiments and Discussions

A. *Experimental Environment*

Python 3.8.8, PyTorch 1.7.1, and Torchvision 0.8.2 were used to train the model. The DPU was created using Vitis-AI 2021.1 and implemented on an Ultra96-v2 board.

TABLE I
Classification accuracy of CIFAR10 using VGG16.

domain	8 classifications	Tail layer training of all weights from random	Tail layer training with fixed existing weights	existing method
ALL	91%	86%	83%	89%
plane	94%	71%	79%	92%
car	98%	84%	84%	93%
bird	86%	87%	89%	85%
cat	84%	83%	78%	88%
deer	93%	91%	75%	91%
dog	83%	86%	76%	88%
frog	98%	94%	95%	92%
horse	91%	92%	86%	93%
ship	−−	98%	88%	97%
truck	−−	83%	81%	92%

TABLE II
Classification accuracy of CIFAR10 using ResNet18.

domain	8 classifications	Tail layer training of all weights from random	Tail layer training with fixed existing weights	existing method
ALL	95%	94%	94%	95%
plane	98%	96%	91%	96%
car	99%	93%	95%	98%
bird	94%	93%	94%	93%
cat	90%	88%	90%	98%
deer	95%	94%	95%	96%
dog	92%	92%	93%	91%
frog	98%	98%	97%	97%
horse	99%	96%	97%	97%
ship	−−	96%	94%	96%
truck	−−	94%	94%	97%

B. *Accuracy of Tail Layer Training Using CIFAR10*

The results of the tail layer training using VGG16 are shown in Table I. Eight classes of VGG16, excluding ship and truck, were trained using the weights learned in ImageNet as initial values, resulting in a 91% correct response rate. Using this 8-class classification model as a base, we trained only the tail layer from initial values given by random numbers using all CIFAR10 images and obtained an overall accuracy rate of 86%. When only the additional classes were trained without updating the weights for the existing classes, an overall correct response rate of 83% was obtained. On the other hand, VGG16, in which the entire model was trained with CIFAR10, had a correct response rate of 89%. Tail layer training shows only three-point accuracy degradation compared to existing methods.

Similarly, the results of tail layer training using ResNet18, ResNet50, and MobileNetV2 are shown in Table II, Table III, and Table IV, respectively. These models achieve recognition accuracies of 95% or better than existing methods. All weights in the tail layer are trained from initial values given by random numbers, and the ex-

TABLE III
Classification accuracy of CIFAR10 using ResNet50.

domain	8 classifications	Tail layer training of all weights from random	Tail layer training with fixed existing weights	existing method
ALL	96%	96%	95%	96%
plane	99%	95%	94%	97%
car	99%	97%	97%	97%
bird	94%	96%	96%	95%
cat	93%	93%	92%	91%
deer	97%	98%	98%	97%
dog	94%	94%	94%	95%
frog	97%	97%	97%	99%
horse	98%	98%	98%	97%
ship	−−	96%	95%	98%
truck	−−	95%	93%	97%

TABLE IV
Classification Accuracy of CIFAR10 Using MobileNetV2.

domain	8 classifications	Tail layer training of all weights from random	Tail layer training with fixed existing weights	existing method
ALL	95%	94%	93%	95%
plane	98%	93%	90%	97%
car	99%	94%	92%	97%
bird	94%	93%	95%	96%
cat	90%	89%	89%	89%
deer	96%	96%	96%	96%
dog	90%	92%	88%	90%
frog	98%	97%	97%	98%
horse	97%	96%	97%	96%
ship	−−	95%	93%	96%
truck	−−	91%	94%	97%

TABLE V
Classification accuracy of CIFAR10 when pretrains with ResNet50 are used as is.

domain	8 classifications	Tail layer training of all weights from random	Tail layer training with fixed existing weights	existing method
ALL	85%	83%	83%	96%
plane	91%	79%	75%	97%
car	96%	89%	88%	97%
bird	76%	77%	75%	95%
cat	75%	75%	73%	91%
deer	83%	75%	86%	97%
dog	85%	84%	84%	95%
frog	90%	88%	90%	99%
horse	88%	86%	84%	97%
ship	−−	91%	91%	98%
truck	−−	89%	88%	97%

isting method was less than one percentage point. It indicates that the higher the accuracy of the existing method, the smaller the accuracy difference with the tail layer training. Similarly, the difference in accuracy between tail layer training, which learns from initial values given by random numbers, and tail layer training with existing weights fixed is also less than 1 point, and the accuracy degradation is sufficiently small.

The results of the same tail layer training experiment are shown in Table V. It decreases in accuracy for all items compared to Table III, which was tested using the same model and data set. It may suggest that when extracting features in layers other than the tail layer, the training results from the existing domain are effective even in the new domain in the additional training.

C. Accuracy of Tail Layer Training Using ImageNet

ResNet50 trained with 999 classes, excluding n15075141 (toilet tissue), which corresponds to the last label in ImageNet, yielded a correct response rate of 74%. Using this model as a base, we trained only the tail layer from initial values given by random numbers using all images in ImageNet. We obtained a correct overall rate of 75% (Fig. 7). On the other hand, the added classes only resulted in a 32% correct response rate. When only the additional classes were trained without updating the weights of the existing classes, the overall correct answer rate was 74%, while the correct answer rate for the added classes was only 44% (Fig. 8). In both cases, the class with the highest error rate was n03887697 (paper towel), which classified 24% when trained from initial values given by random numbers and 12% when trained from additional classes only.

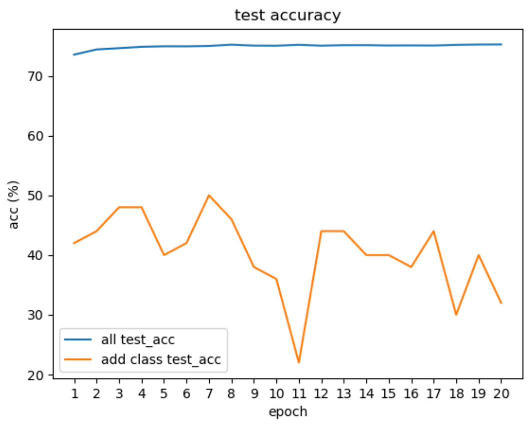

Fig. 7. The Process of Tail Layer Training to Use Randomly Determined Initial Value.

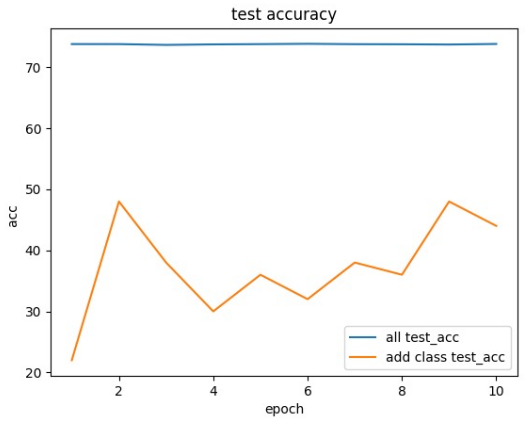

Fig. 8. The Process of Tail Layer Training with Training Only the Additional Classes without Updating the Weights of the Existing Classes.

D. Constraints of Tail Layer Training

Fig. 9 shows the learning process for the tail layer training with only the additional classes performed on VGG16 and CIFAR10 in section B. In addition to the ship and truck being trained here, the accuracy of the plane and car also changed. The 4096-dimensional vectors used as input for the tail layer are converted to two dimensions using principal component analysis (PCA), and the results are shown in Fig. 10. Here, PC1 on the horizontal axis and PC2 on the vertical axis represent the first and second principal components. It can be confirmed that car, truck, ship, and plane are similar among the classes mentioned above. Therefore, it can be inferred that adding a class similar to an existing class will affect the accuracy. Therefore, when performing a multi-class classification on the scale of ImageNet, such as in the Section C clause. In almost all cases, existing classes are similar to the added class, and the misclassification rate increases. However, for a scale of CIFAR10, the number of existing classes is small. The training accuracy of the added classes is

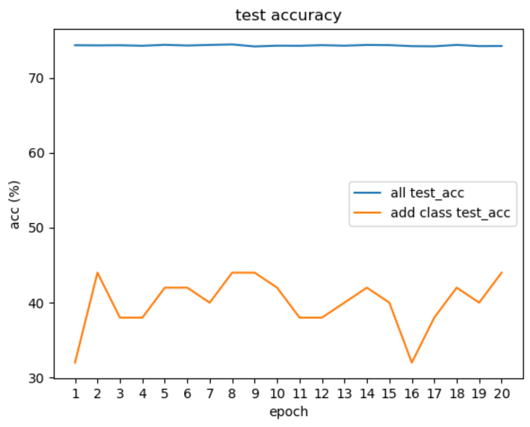

Fig. 9. The Process of Tail Layer Training in which Only Ad

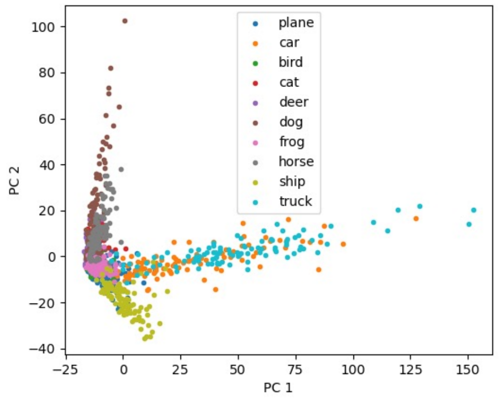

Fig. 10. Result of Compressing The Input to The Tail Layer to Two Dimensions Using PCA.

expected to be equivalent to that of the existing classes.

E. *DPU Modeling and Inference*

The ResNet50 model trained with PyTorch using Vitis-AI can be quantized and compiled for Ultra96-v2 to create xmodel files. Note that VGG16 could not be placed on the DPU because of its large size in this environment. The tutorial provided on DPU on PYNQ[13] includes inference using ResNet50, with a performance of 11.96 FPS. On the other hand, the remainder of ResNet50, excluding the tail fully connected layer, was inferred using DPUs. Only the tail layer was inferred using a model that can be trained with PyTorch, resulting in a performance of 11.42 FPS. Although the tail fully connected layer is implemented with PyTorch on a CPU, the inference speed is sufficient.

V. Conclusion

We trained only the tail layer on the CPU/DPU hybrid system. Using a CNN accelerator allowed faster inference. However, the almost weights were not updated. It separates the tail layer from the CNN accelerator while the CPU is used for training, such as adding classes. However, adding classes similar to existing classes leads to a limited accuracy degradation. We showed that our CPU/DPU hybrid system only had a few performance degradation with a training functionality. Our scheme can be applied to the existing SoC-FPGA-based CNN accelerator.

VI. Acknowledgments

This research is supported in part by the Grants in Aid for Scientistic Research of JSPS and the New Energy and Industrial Technology Development Organization (NEDO). Also, thanks to the Xilinx University Program (XUP), Intel University Program, and the NVIDIA Corp.'s support.

References

[1] PyTorch Research Models. `https://pytorch.org/hub/research-models`.

[2] Yolo. `https://pjreddie.com/darknet/yolo/`.

[3] OpenFace. `https://github.com/cmusatyalab/openface`.

[4] DeepLab. `https://github.com/tensorflow/models/tree/master/research/deeplab`.

[5] OpenPose. `https://github.com/CMU-Perceptual-Computing-Lab/openpose`.

[6] PackNet-SfM. `https://github.com/TRI-ML/packnet-sfm`.

[7] Y. Zhang et al. "Image Super-Resolution Using Very Deep Residual Channel Attention Networks". *ECCV*, 2018.

[8] I. Goodfellow et al. "Generative adversarial nets". *in Proc. Int. Conf. Neural Inf. Process.Syst.Syst.*, 2014.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*, november 1998.

[10] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". `https://arxiv.org/abs/1409.1556v1`.

[11] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". `https://arxiv.org/abs/1512.03385v1`.

[12] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". `https://arxiv.org/abs/1801.04381v3`.

[13] *"DPU on PYNQ"*. `https://github.com/Xilinx/DPU-PYNQ`.