ChangSafe: Embedded System Performance Acceleration and Network Pruning

[1]*Chi-Yu Chang* (張季祐), [1]*Yong-Yi Li* (李詠億), [2]*Shih-Ming Chen* (陳世明), [2]*Shao-Yu Huang*(黃少諭), [1]*Ting Chen* (陳婷), [1]*Chia-Hung Chou* (周家弘), [1]*Chiou-Shann Fuh* (傅楸善)

[1]Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan,
[2] Jeilin Technology Crop. Ltd

b05602049@ntu.edu.tw chensn@jeilin.com.tw wilbert@jeilin.com.tw fuh@csie.ntu.edu.tw

ABSTRACT

Object detection is a common task in computer vision. In recent years, state-of-the-art Convolutional Neural Network (CNN) deep learning methods can provide high accuracy and FPS (Frame Per Second). Before, edge devices are not powerful enough to train and run deep learning models. Therefore, deep learning models are usually run at the cloud: training the deep learning models in a huge data center with powerful computers. Nowadays, with the rise of powerful accelerators such as Graphic Processing Units (GPUs), Tensor Processing Units (TPUs), multi-core processors, models can be deployed on embedded devices with accelerators. Because of embedded device hardware limits, models must be light-weight to achieve low latency time and low memory space. In our implementation, we use network pruning to make our model light-weight. For further improvement, models need to correspond to certain format according to the design of accelerators. In this paper, we will show how to get performance acceleration on embedded device.

Keywords: *Object Detection, CNN, Deep Learning, Network Pruning.*

1. INTRODUCTION

Object detection locate bounding boxes of some certain classes on digital images, such as vehicles, animals, etc. Forming the basis of other computer vision tasks. In the past several years, object detection methods can be approximately divided into two periods, traditional object detection period and deep learning object detection period. Before, Viola Jones detector improve detection dramatically and become one of a classical traditional object detection model. Nowadays, many deep learning-based methods are proposed. From two stages detector to single stage detector. In conclusion, deep learning is a crucial push point enhancing object detection performance.

Over the past several years, an important shift has occurred from cloud-level to device-level processing of artificial intelligence tasks, data and results. Embedded AI is the direct result of this important shift. Traditionally, complex AI computations, such as producing search engine results, were performed at a data center in the cloud. With the implementation of AI models on Graphics Processing Units (GPUs) and Systems on Chips (SoCs), there is less of a dependence on the cloud for AI data processing. Furthermore, with the improvement of accelerators, most calculations are executed on embedded devices. In our application, many embedded devices may be spread over a large region. Therefore, cost should be low. Furthermore, devices may be located in one place with no power supply for a long time. Power consumption is another factor determining how we design our model.

In this paper, we use YOLOV4-tiny as our CNN object detection model. YOLOV4-tiny is the light-weight version of YOLOV4. Number of parameters are much less than YOLOV4. YOLOV4-tiny is capable of multitask, end to end, and multi-scale abilities. At the same time, it can be work in real-time. Hence, we choose YOLOV4-tiny as our original model. YOLOV4-tiny is composed of several CSP blocks, and CSP blocks consist many 3×3 convolutional operations. However, the number of our embedded device's acceleration units are multiple of sixteen. In model acceleration, we replace 3x3 convolutional kernels with 4×4 convolutional kernels. With this improvement, we decrease our model inference time, and reduce power consumption. After we change YOLOV4-tiny structure from 3×3 convolutional operations to 4×4 convolutional operations. We considered that there may be some useless information exist in our model. Next, we have to truncate information in order to reach our goals of low power and memory consumption. Model compression is a necessary work to do. Compression is an important part in our application. There are many model compression types, like model pruning, knowledge distillation, and parameter quantization. Model pruning removes unimportant

weights from original model to reduce model size and computation capacity. Model pruning can be categorized into fine-grained methods and coarse-grained methods. Fine-grained methods prune model in weight-level and can reach high compression rate. Considering hardware limitation, Fine-grained methods require special hardware support. Coarse-grained methods can be implemented in channel-level. Comparing to fine-grained method, Coarse-grained method is a more general solution, but the precision may be lower than fine-grained method. Knowledge distillation uses teacher student mechanism to learn important knowledge from teacher large pre-trained model. However, there are more constraints than model pruning. We choose model pruning method consequently. Parameter quantization change datatype of weights. Usually, weights are saved with floating point data type. Turning floating point into integer reduce model size. Before deploy our model on embedded device, we also apply parameter quantization method to change data type of weights.

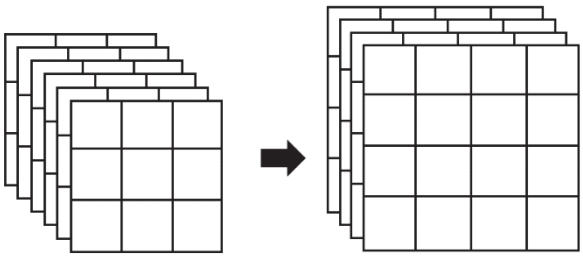


Fig. 1. 3×3 convolution to 4×4 convolution.

2. METHOD

In our implementation, we first change our model structure. Because our embedded device performs better in 4×4 convolution operation than in 3×3 convolution operation, we replace most 3×3 convolutional kernels with 4×4 convolutional kernels. In order to strike the balance between inference time and model precision, removing useless information in our model is an important task to do. Then, we use network pruning technique to reduce model size in secondary step.

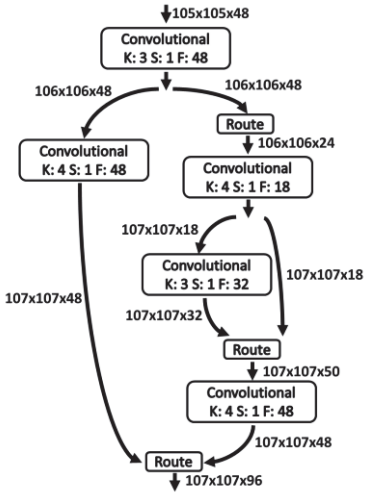


Fig. 2. CSP block in our design. K denotes kernel size. S denotes stride. F denotes number of filters. In each convolutional layer, we apply leaky ReLU activation function and batch normalization.

2.1. Network Redesign

When we design a new 4x4 convolution structure to replace original YOLOV4-tiny, we try to keep number of parameters the same in each layer. However, it is hard to make number of parameters in each layer exactly equal, and we finally control these errors in a reasonable range.

YOLOV4-tiny structure can be decomposed into one input block, three CSP blocks, and two output blocks. We follow similar concept and make sure that parameters in new design do not exceed original version 20% in each block.

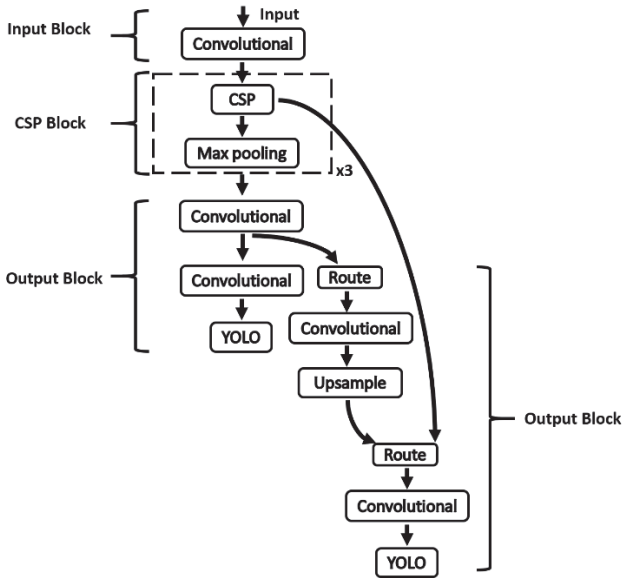


Fig. 3. Entire structure of our new design network. Convolutional layer may consist of several continuous convolutional layers. YOLO layer denotes prediction layer in YOLOV4-tiny.

2.2. Network Pruning

Here, we use fine-grained model pruning method to compress our redesign model from previous section to reach our goals of low power consumption and low memory requirement.

2.2.1. Sparse Training

Due to the existence of unimportant weights, we need to find solutions to solve this problem. An intuitive approach is using zero masks to deactivate near zero weights. However, the rum-time memory saving is limited using this approach because more memory space is used by zero masks. Furthermore, special hardware and libraries are needed to support this approach.

We turn to channel-level sparsity method to eliminate this drawback. This approach does not increase any additional parameters to model and memory during run-time. Batch normalization is a technique commonly used in deep learning to prevent gradient vanishing. Normalizing mini batch data before activation function is the common practice.

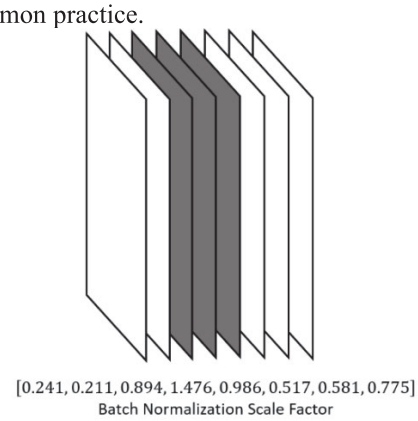


Fig. 4. One channel corresponds to one scale parameter. If the scale parameter is large, then the channel is important, like gray channels above.

Batch normalize functions are defined in equation (1) and (2). Let μ_B and σ_B be mean and standard deviation over mini batch data B. γ and β are trainable parameters for scale and shift separately. x_i denotes original values. $\hat{x_i}$ denotes standardized values. y_i denotes output values of batch normalization.

$$\hat{x_i} = \frac{x_i - \mu_B}{\sigma_B} \tag{1}$$

$$y_i = \gamma \hat{x_i} + \beta \tag{2}$$

Scale parameter γ becomes a value indicating channel importance: the higher, the more important. Sparse training is different from normal training. In normal training, we expect correct classification, accurate localization regression, and trainable channel parameters. In contrast, sparse training may try to find out whether a channel is important or not. Sparse training loss is defined as:

$$L = \sum l(\hat{y}, y) + \lambda \sum_{\gamma \in \Gamma} g(\gamma) \tag{3}$$

where $\hat{y}$ and y are predictions and ground-truths; γ denotes scale parameters in batch normalization layers.

2.2.2. Channel Pruning

According to sparsity training, we know about the importance of channels. Removing low sparsity score layer results in smaller model size and moderate accuracy loss. We can prune these layers with several approaches. **Prune top p lowest scores**. A simple solution is using sorted sparsity score, and prune the lowest R% layers. Before we take this approach, it needs to be guaranteed that model is well trained in sparsity training stage. To ensure the network is well trained in sparsity training, it is recommended training network with long steps.
After pruning, network performance will drop critically. Finetune training can solve this problem. To avoid pruning many layers resulting in performance drop, we apply iterative pruning strategy. With iterative pruning strategy, we only prune small percentage of layers iteratively. Network pruning flowchart is showed in Figure 5.

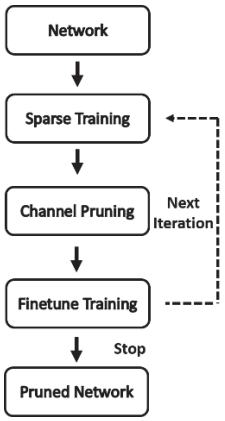


Fig. 5. Network pruning flowchart.

2.3. Memory Pipeline

In order to use memory efficiently during run time, several memory segments are spilt in SRAM. For example, suppose input size is $320 \times 224 \times 3$ and one memory segment can be set as multiple of 80 x 56. Input and output of layers are saved in memory segments. As forward passing, memory segments an be further split into smaller parts. Memory pipeline is showed in Fig 6.

Layer \ Memory	80×56×32	80×56×32	80×56×32	80×56×32	80×56×32	80×56×32
Convolution	L0_out	L0_out	L0_out	L0_out		
Convolution	L1_in	L1_in	L1_in	L1_in	L1_out	L1_out
Convolution	L2_out	L2_out			L2_in	L2_in
Route	L3_in	L3_in/out				
Convolution		L4_in				L4_out
Convolution					L5_out	L5_in
Route					L6_in/out	L6_in/out
Convolution			L7_out	L7_out	L7_in	L7_in
Route	L8_in/out	L8_in/out	L8_in/out	L8_in/out		
Max_Pooling	L9_in	L9_in	L9_in	L9_in	L9_out	
Convolution	L10_out				L10_in	

Fig. 6. Memory Pipeline. In this example, six memory segments of size 80 x 56 x32 are used during model forwarding. Input and output of each layer can be placed in these memory segments. Light color indicates output of layer and deep color indicates input of layers.

2.4. Parameter Quantization

Here we use affine quantization method. $f(\cdot)$ is a transition function from float type to integer type and is defined as:

$$f(x) = m \cdot x + n \tag{4}$$

where $m = \frac{2^b-1}{f_{max}-f_{min}}$, $n = round(f_{max} \cdot f_{min}) - 2^{b-1}$, b is number of bits of an integer, f_{max} is maximum value of weights in float type, f_{min} is minimum value of weights in float type.

Quantization function transforms data type form float to integer type and transformed values beyond range must be clipped. Dequantization function is the inverse function of quantization function. These two functions satisfying $DQ(Q(x)) \approx x$.

$$Q(x) = \begin{cases} max(round(f(x)), 127), & f(x) \geq 0 \\ min(round(f(x)), -128), & f(x) < 0 \end{cases} \tag{5}$$

$$DQ(x) = \frac{(x-n)}{m} \tag{6}$$

$g_i(\cdot)$ is the transformation function of layer i, K_i is the set of kernels in layer i, and I_i is the input values of convolutional layer i. O_i is the computed values of convolutional layer i. $c(\cdot)$ is the compress function and $dc(\cdot)$ is the decompress function. $bias$ is 64 here.

$$c(x) = x + bias \tag{7}$$

$$dc(x) = x - bias \tag{8}$$

$$g_0(I_0) = c(Leaky\ ReLU(I_0 \cdot Q(K_0))) \tag{9}$$

$$g_i(I_i) = c(Leaky\ ReLU(dc(I_i) \cdot Q(K_i))) \tag{10}$$

Compress and decompress can be done using shifting. Compressed values ranging from 0 to 255 can be saved with 8 bits integer, and uncompressed values ranging from -64 to 191 must be saved in 9 bits signed integer. Using compress technique can save half SRAM space. Data transfer flowchart is showed in Fig 7.

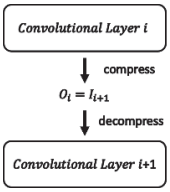
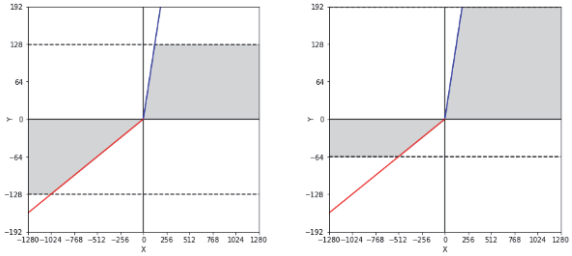


Fig. 7. Data transfer between convolutional layers.

Considering leaky ReLU function values, range size of negative values is usually smaller than positive values. We store leaky ReLU values in 8 bits data type ranging from -128 ~127, and the range size of positive and negative values are nearly equal. Therefore, we shift values to -64 ~191. Additionally, we can compress and decompress data by shifting.



(a) (b)

Fig. 8. (a) is signed 8 bits range leaky ReLU values. (b) is shifted signed 9 bits range leaky ReLU values.

2.5. YOLO Layer Quantization

YOLO layer is a special type of layer in YOLOV4 tiny. In order to save memory space, shrinking output values from YOLO layer. In YOLO layer, 25 values are computed on a center. 25 values are composed of 21 logistic values, 2 location values, and width and height values. Normal function $N(\cdot)$ is used for location values of bounding boxes. Logistic function $L(\cdot)$ is a function for turning integer size from 8bits to 5 bits and it is used for 21 logistic values. Scaled logistic function $SL(\cdot)$ is used for width and height values of bounding boxes. Scaled logistic function $SL(\cdot)$ is similar to logistic function $L(\cdot)$. The only difference is that scale and shift variables are added in scaled logistic function $SL(\cdot)$.

$$N(x) = \begin{cases} max(round\ (t(x))), 127), & t(x) \geq 0 \\ min(round\ (t(x))), -128), & t(x) < 0 \end{cases} \tag{11}$$

$$L(x) = round\ \left(\mu \frac{1}{1+e^{-(x-192)}}\right) \tag{12}$$

$$SL(x) = round\ \left(\mu\left(\alpha \frac{1}{1+e^{-(x-192)}} + \beta\right)\right) \tag{13}$$

where $t(x) = \mu(x-192)$, $\alpha = \frac{67}{64}$, $\beta = -\frac{3}{128}$.

Here, we prepared 3 lookup tables for 3 different functions. Using lookup table can save computation resource by sacrificing little memory space.

2.6. Advanced ALU Operator

Traditional ALU (Arithmetic Logic Unit) operator can do add, multiply, and shift operations. But only one type of operation can be done by ALU at one time. Advance ALU operator introduces MAC (Multiply Accumulate) operation, multiply and add operations can be completed at the same time. However, this change makes the operations flow be less flexible than traditional ALU operator. The combination of MAC and shift operations can further extend new operations, like MAS (Multiply Accumulate Shift), MSA (Multiply Shift Accumulate). These new operations only influence dataflow and won't increase hardware cost of ALU. ALU also benefits from make circuit usage higher. Thus, SuperALU applies new operations and MSAS operation. MSAS (Multiply Shift Accumulate Shift) operation is a special design for convolutional operation in CNN network corresponding to quantization method of our

model. Define register r_1, r_2, r_3, r_4. MAC operation is listed below.

$$MAC: r_4 = r_1 + r_2 \times r_3 \quad (9)$$

$$MAS: r_5 = (r_1 + r_2 \times r_3)\ shift\ r_4 \quad (10)$$

$$MSA: r_5 = r_1 + (r_2 \times r_3)\ shift\ r_4 \quad (11)$$

$$MSAS: r_5 = r_1 + ((r_2 - 64) \times r_3)\ shift\ r_4 \quad (12)$$

3. RESULTS

In this section, we focus on how model change influences model memory requirements. Number of parameters becomes a crucial standard determining whether a model is efficient or not. Small number of parameters and suitable data format corresponding to hardware design can help us improve performance.

3.1. Evaluation

Object detection usually includes two subtasks: localization regression and classification. In detection, IOU (Intersection Over Union) is a crucial concept. IOU helps us understand the level of overlapped area between predicted bounding box and ground-truth bounding box. IOU threshold is another factor determining how we evaluate our model. When the IOU score is higher than IOU threshold, we take the prediction as correct. Based on IOU, we can compute precision and recall. Furthermore, we use mean Average Precision (mAP) as our main object detection evaluation metric. We take benchmark 1 and benchmark 2 as our test images. In benchmark 1, the detection difficulty is that the overlapped area of the bicycle and the dog. And in benchmark 2, the difficulty is that it is hard to detect the horse correctly.

3.2. Dataset

We use PASCAL (Pattern Analysis, Statistical modelling and ComputAtional Learning) Visual Object Classes (VOC) dataset and Common Object in Context (COCO) dataset in this paper. PASCAL VOC and COCO datasets contain 20 and 80 different classes, respectively. COCO dataset is much bigger than PASCAL VOC dataset. Thus, we use COCO dataset as our pre-trained dataset. Pretraining helps us perform better on PASCAL VOC dataset.

3.3. Leaky ReLU

Original leaky ReLU (Rectified Linear Unit) is defined below. λ denotes the slope coefficient for negative values.

$$Leaky\ ReLU(x) = \begin{cases} x, & x \geq 0 \\ \lambda x, & x < 0 \end{cases} \quad (13)$$

The standard value of λ is 0.1. However, acceleration can be improved, if we change the constant to 0.125 considering hardware device properties (right shift 3 bits). The division operation can be replaced with shift operation.

Here, we set two different input sizes 416×416×3 and 320×224×3, and two types leaky ReLU: $\lambda = 0.1$ and $\lambda = 0.125$. The combination of these two pairs setting may slightly make mean Average Precision (mAP) different. Table 1 and Fig. 10 show results of our experiment.

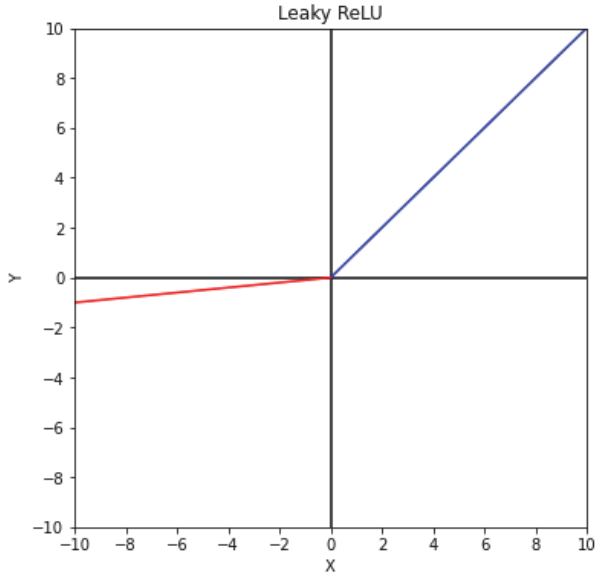
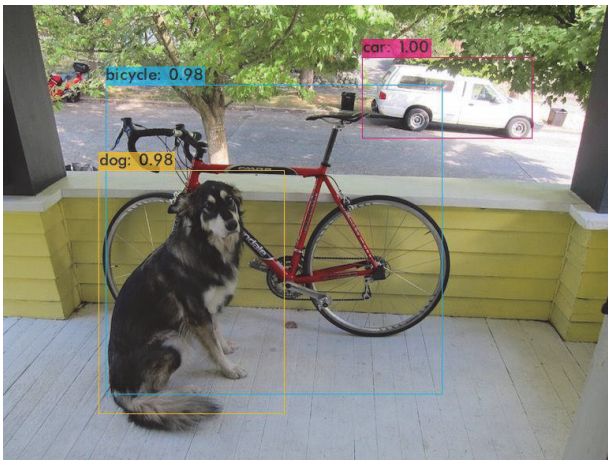


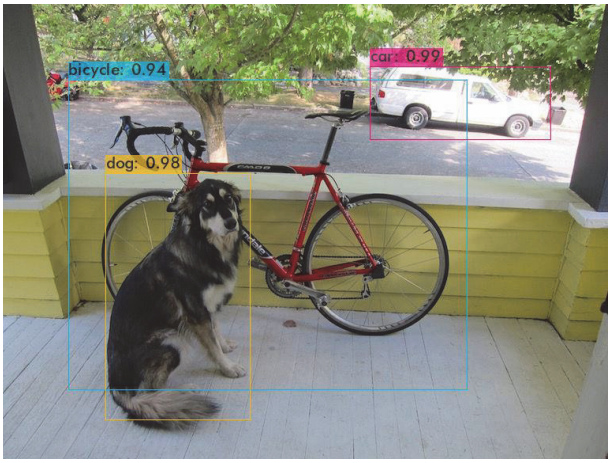
Fig. 9. Leaky ReLU

Table 1. mAP comparison. Changing λ from 0.1 to 0.125 does not make performance drop obviously. Input size is a crucial factor impacting performance. Large image size give model more information, and therefore make performance better.

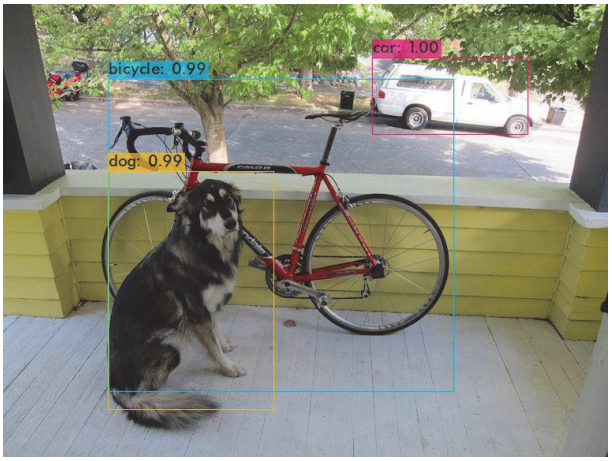
λ	Input size	mAP
0.1	320×224×3	66.5
0.125	320×224×3	66.6
0.1	416×416×3	73.1
0.125	416×416×3	72.7



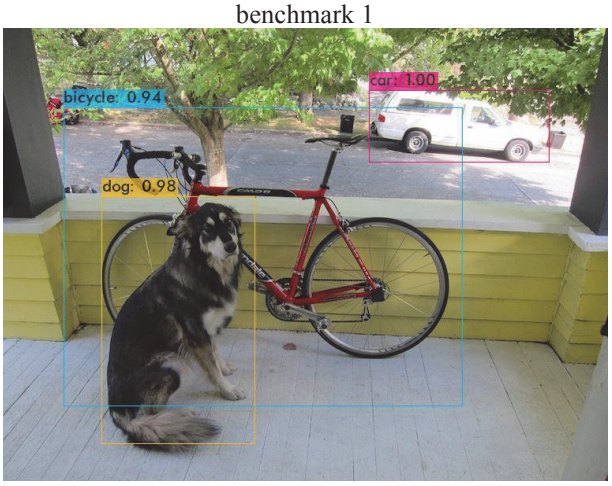
(a) $\lambda = 0.1$, input size = 320×224×3
benchmark 1



(b) $\lambda = 0.1$, input size = 416×416×3 benchmark 1



(c) $\lambda = 0.125$, input size = 320×224×3 benchmark 1

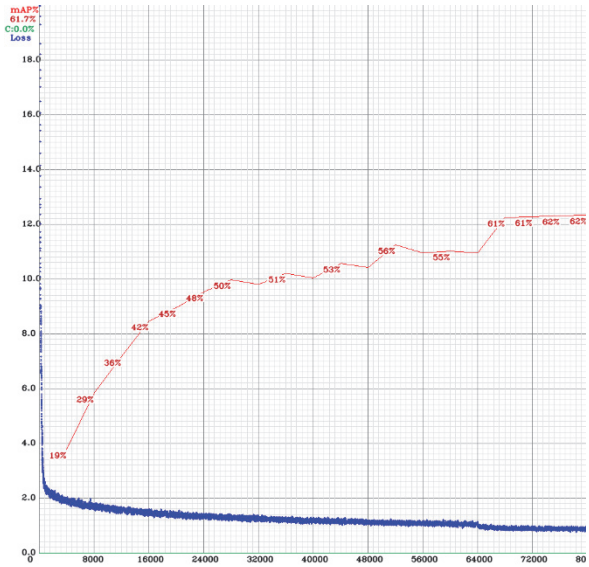


(d) $\lambda = 0.125$, input size = 416×416×3 benchmark 1

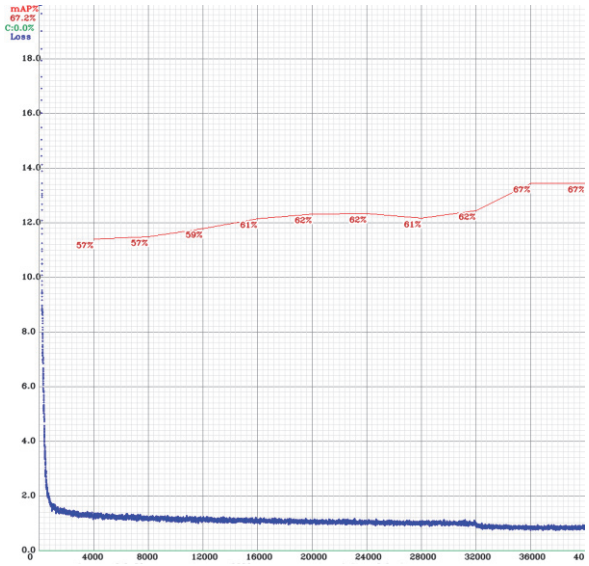
Fig. 10. Benchmark 1 comparison. In benchmark 1 above, we can observe performance with $\lambda = 0.125$ is slightly worse than that with $\lambda = 0.1$. The tradeoff between the drop of performance and acceleration can be accepted.

3.4. Pre-Trained Importance

Training with pre-trained weights has two major advantages, making training faster and performance better. With pre-trained weights, tasks of model have been completed roughly, and we can only finetune pre-trained weights to achieve higher performance on our own dataset. Pre-trained weights are usually trained with large dataset with labeled data. Hence, it is helpful for model to escape from local optimal. We can observe difference of YOLOV4-tiny training in Fig. 11.



(a) YOLOV4-tiny training without pre-training.



(b) YOLOV4-tiny training with pre-training.

Fig. 11. YOLOV4-tiny training. In (a), loss drop slowly and the oscillation of loss is occurred as training progressing. In contrast, loss of (b) is converged in few steps and seems more stable than loss of (a). Overall, Training with pre-training is better.

3.5. Pruning results

In YOLOV4 tiny, there are five types of layer, including convolutional, route, max pooling, upsample, and yolo layer. Route layer combines outputs of previous layers and determine which part should be passed to next layer. YOLO layer is capable of doing prediction. Most calculations and trainable parameters are in convolutional layers. Thus, we only prune convolutional layers to decrease number of parameters. Numbers of parameters in YOLOV4-tiny with different λ and input size after iterative pruning are showed in Table 4. Pruning ratio is 0.15 and pruning iteration is 2.

Table 2. Numbers of parameters of pruned YOLOV4-tiny.

λ	Input Size	Parameters
0.125	320x224x3	5,122,809
0.125	416x416x3	4,295,292
0.1	320x224x3	-
0.1	416x416x3	3,590,386

Table 3. Redesigned network pruning.

Layer index	Total channels	First Pruning	Second Pruning
0	24	24	24
1	48	48	48
2	48	48	48
3	18	18	18
4	32	32	32
5	48	48	48
6	96	96	96
7	36	36	36
8	64	64	64
9	96	94	94
10	192	186	186
11	72	47	47
12	128	128	97
13	192	141	94
14	512	419	375
15	256	205	171
16	384	347	199
17	192	91	84

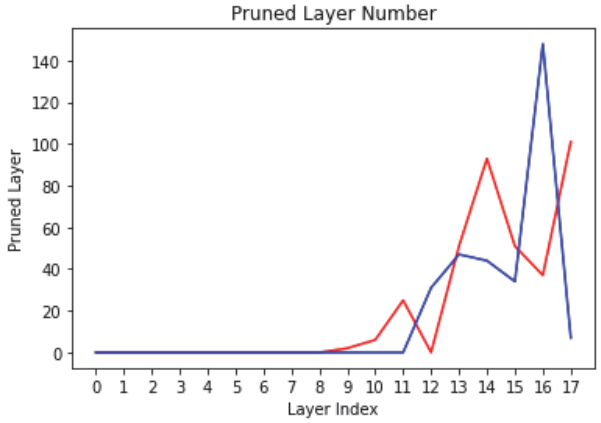


Fig. 12. Pruned layer number distribution in Table 3. We can find that most pruned layers belong to the output blocks. Low level feature extraction is so important that front layer do not be pruned.

3.6. Quantization results

In this experiment, we quantize original YOLOV4-tiny network. We implemented several different data type versions, including 8 bits integer, 16 bits integer, and unsigned mixed 8 bits integer. In our experiments, we change m value of function (4) in each version. Class scores of benchmark 1 are listed in Tables 4, 5, and 6. Class scores of benchmark 2 are listed in Tables 7, 8, and 9.

Table 4. 8 bits scores benchmark 1

m	class	score
8	Bicycle	0
	Dog	37
	Car	70
16	Bicycle	44
	Dog	67
	Car	91
32	Bicycle	44
	Dog	67
	Car	91

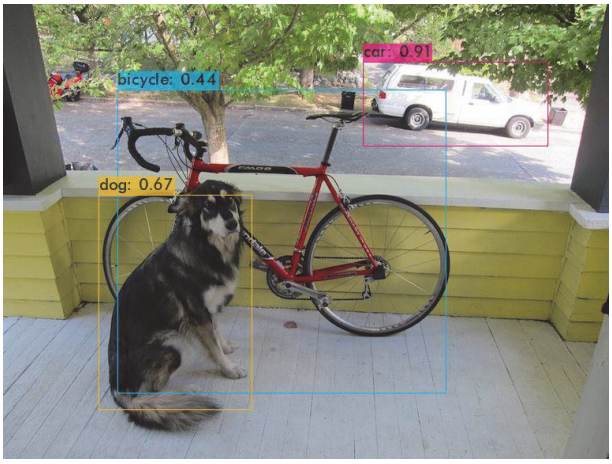


Fig. 13. 8 bits (m = 16) benchmark 1

Table 5. 16 bits scores benchmark 1

m	class	score
8	Bicycle	28
	Dog	32
	Car	67
16	Bicycle	55
	Dog	70
	Car	93
32	Bicycle	66
	Dog	82
	Car	97
64	Bicycle	64

	Dog	86
	Car	98
128	Bicycle	61
	Dog	88
	Car	99
256	Bicycle	61
	Dog	88
	Car	99
512	Bicycle	61
	Dog	89
	Car	99
1024	Bicycle	61
	Dog	89
	Car	99

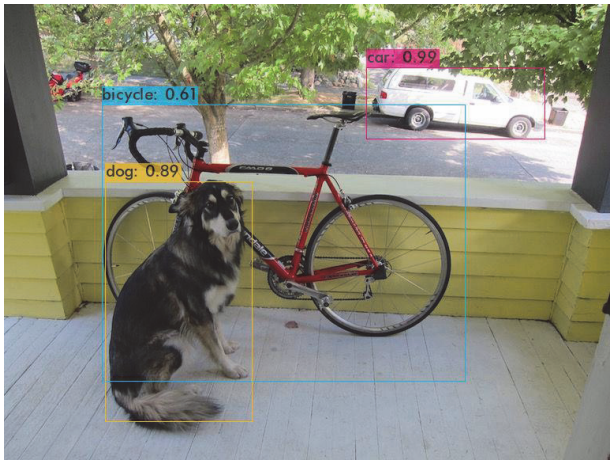


Fig. 14. 16 bits (m = 1024)
benchmark 1

Table 6. unsigned mixed 8 bits scores
benchmark 1

m	class	score
16	Bicycle	78
	Dog	97
	Car	100
32	Bicycle	27
	Dog	93
	Car	26

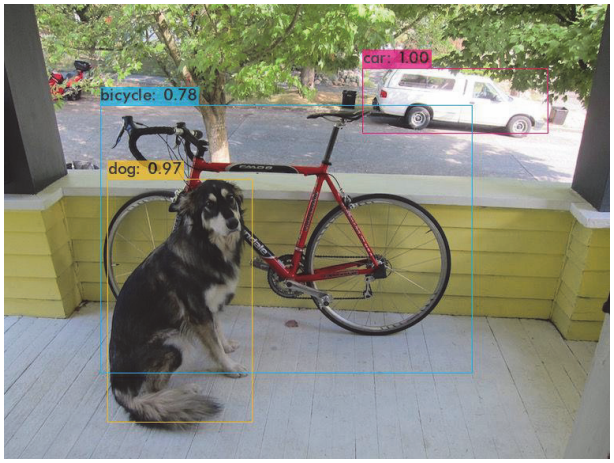


Fig. 15. unsigned mixed 8 bits (m = 16)
benchmark 1

Table 7. 8 bits scores
benchmark 2

m	class	score
8	Person	43
	Dog	-
	Horse	-
16	Person	72
	Dog	36
	Horse	
32	Person	80
	Dog	38
	Horse	63

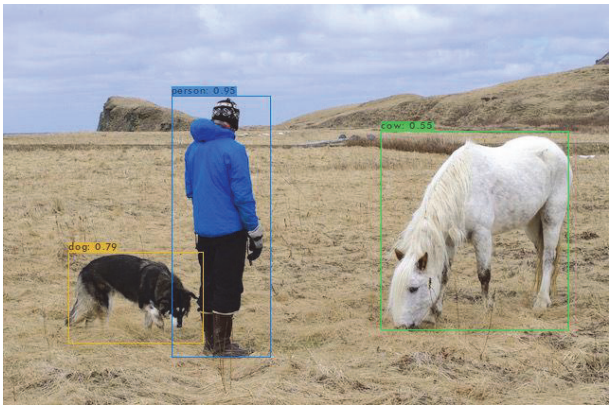


Fig. 16. 8 bits (m = 16)
benchmark 2

Table 8. 16 bits scores
benchmark 2

m	class	score
8	Person	43
	Dog	-
	Horse	-
16	Person	82
	Dog	38
	Horse	-
32	Person	90
	Dog	63
	Horse	-
64	Person	93
	Dog	72
	Horse	-
128	Person	94
	Dog	76
	Horse	-
256	Person	94
	Dog	78
	Horse	-
512	Person	95
	Dog	79
	Horse	-
1024	Person	95
	Dog	79
	Horse	-

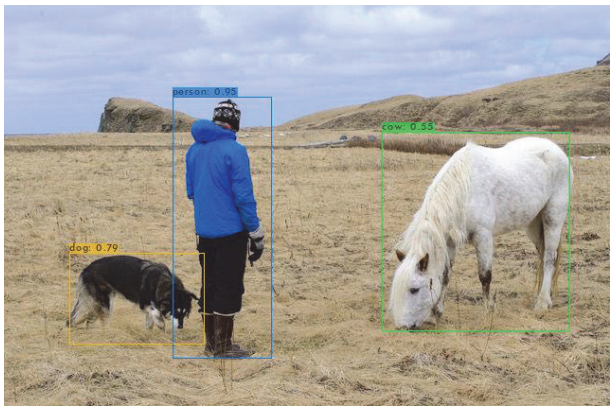


Fig. 17. 16 bits (m = 1024)
benchmark 2

Table 9. unsigned mixed 8 bits scores
benchmark 2

m	class	score
16	Person	98
	Dog	92
	Horse	-
32	Person	65
	Dog	89
	Horse	72

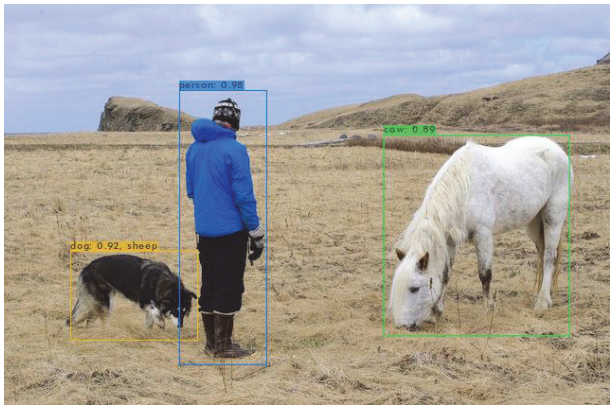


Fig. 18. unsigned mixed 8 bits (m = 16)
benchmark 2

4. CONCLUSIONS

Nowadays, numbers of embedded applications rise dramatically. Artificial intelligence is a majority of these application. Low power consumption and low hardware cost are key points to develop embedded AI system. The tradeoff between performance and hardware cost must be measured according to various situations. Now, we already have a moderate performance model. How to improve it is a challenge to be overcome in the future. When we get incoming data with working device, we can use these extra data improving our model. Moreover, we can implement IoT (Internet of Things) technique to help us communicate with devices in real time.

ACKNOWLEDGMENT

This research was supported by Jeilin Technology.

Thanks to Jeilin Technology for providing the experiment data to this paper.

REFERENCES

[1] Z. C. Jiang, L. Q. Zhao, S. Y. Li, and Y. F. Jia, “Real-Time Object Detection Method for Embedded Devices,” *arXiv:2011.04244v2*, 2020.

[2] Z. Liu, J. Li, and Z. Shen et al., “Learning Efficient Convolutional Networks through Network Slimming,” *Proceedings of IEEE International Conference on Computer Vision,* Venice, Italy, pp. 2736-2744, 2017.