Research on Model Compression for Embedded Platform through Quantization and Pruning

Xiao Hu¹,a , Hao Wen²,b
¹Department of Electrical & Computer Engineering, University of Florida, Gainesville, Florida, 32611, USA
²Department of Electrical & Computer Engineering, University of Florida, Gainesville, Florida, 32611, USA
a-e-mail: xiao.hu@ufl.edu, b-e-mail: h.wen@ufl.edu

Abstract. So far, artificial intelligence has gone through decades of development. Although artificial intelligence technology is not yet mature, it has already been applied in many walks of life. With the explosion of IoT technology in 2019, artificial intelligence has ushered in a new climax. It can be said that the development of IoT technology has led to the development of artificial intelligence once again. But the traditional deep learning model is very complex and redundant. The hardware environment of IoT cannot afford the time and resources cost by the model which runs on the GPU originally, so model compression without decreasing accuracy rate so much is applicable in this situation. In this paper, we experimented with using two tricks for model compression: Pruning and Quantization. By utilizing these methods, we got a remarkable improvement in model simplification while retaining a relatively close accuracy.

1. Introduction

This paper mainly focuses on two model compression methods: quantization and pruning. The idea of model compression leads to the concept of embedded AI, a technical concept that allows smaller and faster AI algorithms models deployment on IoT devices and to run on terminal systems. Embedded Artificial Intelligence can be applied in various vertical fields of industry and has great potential in miniature electronic devices. Nowadays, deep learning models are getting deeper and more complex in structure. The latest language model like GPT-3[6] has become so gigantic for any researchers to use, which contains 175 billion parameters, making it 17 times as large as its previous model GPT-2[7]. On the one hand, this trend eventually hinders on-device Deep Learning applications from use, especially in the combination of embedded systems and artificial intelligence. On the other hand, the calculation speed and accuracy of the transplanted models are usually not equivalent to that of PC.

Ever since Convolutional Neural Network (CNN)[1] was first introduced in 2012 for the ImageNet classification task, it has been promoted and widely used as benchmarks for different tasks: sentiment analysis, text understanding, etc[3]. The excellent performance of CNN benefits from its millions of trainable parameters. These parameters and structured information need to be stored on the hard disk and then reloaded in memory during inference. The intermediate activation values of CNN are also larger than that of what has been stored as model parameters in hard disks. It becomes a dilemma for IoT devices since the larger size of the Deep Learning Model means a great burden for embedded devices. At this stage, the mainstream ARM architecture processor is not fast enough. If a large
number of image calculations are to be performed, it may even take a few minutes to process a single image on a large CNN network on an embedded device[2]. Therefore, in order to solve such problems, our experiments utilize quantization and pruning to optimize 2 trending CNN-like structure models, so that the optimized lightweight model can be transplanted to the System on Chips (SoC) platform for reasoning. Furthermore, experiments on INT8 quantization also proved its superiority. Quantization accelerates the inference of the AlexNet model. Compared to the same models running on PC, our final realization achieved a relatively close accuracy on CIFAR10 and CIFAR100, whose accuracy was reduced by 0.08 in the case of reducing the inference time by 1/3 of the non-quantized models. By pruning the YOLO-v3 model through introducing scaling factors in the Batch Normalization layer, our reasoning time of each frame obtained from the class prediction and object recognition for the same video has been reduced, and finally, we even obtained an overall improved accuracy in each frame.

2. Quantization

Quantization has been proved its efficiency in signal processing, image processing, etc. And it is widely applied in the communication field. In order to perform faster forward and back-propagation computation in Deep Neural Networks and stores tensor data at a lower cost. A quantization technique: lowering the bit-width of models by quantizing Floating Point 32bits (FP32) into Integer 8bits (INT8) is proposed by Google [4]. In the competition of LSVRC-2012, AlexNet achieved an unprecedented low top-5 error rate at 15.3% on ImageNet. It uses a structure of 600 million parameters and over 650,000 neurons including 5 convolutional layers and 3 fully connected layers. But it still lacks speed in practice when the actual computational resource is limited. Hence in this chapter, we first discuss the structure of a standard CNN benchmark: AlexNet, and then we will use a matrix multiplication example to illustrates our quantization scheme.

2.1. CONVOLUTIONAL NEURAL NETWORK: ALEXNET

Comprising of three major components: convolutional Layer, pooling Layer, and fully connected Layer, AlexNet uses 5 major convolutional layers and 3 fully connected layers design in Fig. 1.

![FIGURE 1. Structure of AlexNet](image)

It is allowed to focus local features like edges and line components through Convolutional kernels shown in Fig. 2 while extracting higher-level information and avoid overfitting from max-pooling as shown in Fig. 3. These prior components somehow function as pruning since they are using less parameters to prevent overfitting in the latter component: fully connected layers.
FIGURE 2. Convolutional layer focuses on edges and extracting features

FIGURE 3. Max-pooling layer eliminates extra dimensions and prevent over-fitting

Hence, fully connected layers are then able to be trainable with low-dimensional feature maps extracted from lower layers. However, when backpropagation uses a stochastic gradient descent in a batch size of 128 using the following weight updating equation:

\[ v_{i+1} = 0.9 \cdot v_i - 0.0005 \xi \cdot \omega_i - \xi \cdot \left( \frac{\partial L}{\partial \omega} \right) \cdot D_i \]  

\[ w_{i+1} = w_i + v_{i+1} \]  

We can see multiple floating-point arithmetic calculations are involving. And these floating-point methods will complicate the process of training and cost a lot of time and hardware resources, that is very inappropriate for embedded platform.

2.2. QUANTIZATION METHOD
Quantization is very common for image processing, the common trick is quantifying a Float32 type of image to an INT8 image for compression storage. Since most of the quantization in the industry is linear quantization instead of non-linear methods, so it can be roughly regarded as a linear transform between matrices in real applications. Since results from convolutional layers and fully connected layers are actually performing a series of matrices computations. When given a tensor \( r \), we can quantize with the following equation:
tensor \( q = \text{round} \left( \frac{1}{s} \cdot \text{clip}(r, -a, a) \right) \)  

where \( s \) is the scaling factor, and \( a \) is the threshold, and \( \text{round}(\ast) \) is a rounding function, and \( \text{clip}(r; -a; a) \) is defined as followed:

\[
\text{clip}(q) = \begin{cases} 
-a, r \in (-\infty, -a) \\
0, r \in [-a, a] \\
a, r \in (a, \infty)
\end{cases}
\]  

For instance, we are scaling a matrix multiplication process in a fully connected layer or convolutional layer. Let \( R_1 \) and \( R_2 \) be the Floating-point matrix in size of \( N \times N \) where \( N \) is the dimensions of each matrix and assume that \( R_3 \) is the multiplication of \( R_1 \) and \( R_2 \). Hence: we have

\[ R_3 = R_1 \cdot R_2 \]  

And let \( R_{1}^{ij}, R_{2}^{ij}, R_{3}^{ij} \) denotes the element of each matrix respectively. We then can further apply the scaling factor \( s_1, s_2, s_3, \) and zero-point \( z_1, z_2, z_3 \) into Equation (5), and it can be rewritten as follows:

\[ s_3(Q_3^{jk} - z_3) = \sum_{j=1}^{N} s_1 s_2 (Q_1^{ij} - z_2) (Q_2^{jk} - z_3) \]  

In Equation (6): \( Q_{1}^{ij}, Q_{2}^{ij}, Q_{3}^{ij} \) respectively represents quantized INT8 element from \( R_{1}^{ij}, R_{2}^{ij}, R_{3}^{ij} \). For simplification purposes inspired by [4], an empirical solution requires us to set \( M = \frac{s_1 s_2}{s_3} \) and \( M \) can be easily computed by \( M = 2^{-n} M_0 \) through bit-wise arithmetic shift operations, where \( M_0 \) is a fixed point value, (i.e. the number of digits after the decimal point is fixed). Subsequently, we can transform both the convolutional layer and fully connected layer into the similar quantization forms shown below:

\[ Q_3^{jk} = M_{\text{layer}} \cdot \sum_{j=1}^{N} (Q_1^{ij} - z_2) (Q_2^{jk} - z_3) + z_1 \]  

where \( M_{\text{layer}} \) is dependent on information of each layer’s weights. In other words, it can be computed by obtaining the scale and zero points from the min and max values in each layer. Once all the weights are quantized into INT8 values, thus the quantization process is completed.

2.3. IMPLEMENTATION AND EVALUATION OF QUANTIZATION

To leverage the effectiveness of quantization during preprocessing pictures, enlightened by [5], in this paper, we managed to quantize most Floating Point 32bits (FP32) parameters in AlexNet into Integer 8bits (INT8) and preserved the accuracy of benchmarks when transplanting our model into a Jetson NANO platform with 256 cores of GPU. The following Table 1 and Table 2 demonstrate the supremacy of the Quantization method in reducing calculation and inference latency-vs-accuracy loss. Less than 0.09% accuracy was lost with a speedup of 1.48 on the CIFAR-10 dataset. And the top-5 accuracy of the INT-8 model also showcases an improvement in a speedup of 1.34.
TABLE 1. Object Classification speed and Top-5 accuracy on CIFAR-10 of Float32 and INT8 models, Latency is measured on Multi-cores Jetson NANO

| Training Type | Top-5 Accuracy | Latency (ms) |
|---------------|----------------|--------------|
| Float32       | 82.57          | 31.30        |
| INT8          | 82.49          | 21.14        |

TABLE 2. Object Classification speed and Top-5 accuracy on CIFAR-100 of Float32 and INT8 models, Latency is measured on Multi-cores Jetson NANO

| Training Type | Top-5 Accuracy | Latency (ms) |
|---------------|----------------|--------------|
| Float32       | 81.84          | 40.32        |
| INT8          | 81.77          | 30.17        |

3. Network pruning

Pruning approaches are commonly classified into 2 major categories:

• Pruning on well-trained models and generating new light-weighted models.
• Constructing a self-pruning model through the training process.

However, in actual practice: the latter approach often requires more time to train and may obtain lower accuracy, even it is a worthwhile attempt shown in [17]. So in this paper, we provide an implementation experiment by pruning a well-trained model and tested it on the Microsoft-COCO dataset[13]. Since much research do not deliver deployment on small scale GPU after pruning. As a matter of fact, we also examine how batch-normalization pruning helps improve the accuracy of the original model on Jetson NANO[18].

FIGURE 4. Structure of YOLO-v3

3.1. PRUNING EXPERIMENT ON YOLO-V3

We conducted a pruning experiment on the latest YOLOv3 model. You only look once Version Three (YOLO-v3) is now the state-of-the-art real-time object detection system[14]. On a Pascal Titan X, it processes images at 30 FPS and has a MAP of 57.9% on COCO test-dev[20]. Three fundamental blocks: Convolutional layer, Batch-Normalization (BN) layer, and a Leaky-ReLU layer constitutes DarknetConvBatchNormalization-LeakyRelu Block (DBL). And DBL is the fundamental component
of YOLO-v3 as indicated in Fig. 4, which makes it more resizeable in pruning since it has abandoned the Max-pooling layer.

3.2. PRUNING SCHEME

Even in the process of training, the distribution of inputs from the hidden layers can be always changing and may lead to a vanishing gradient problem. Thus, the Batch-Normalization layer is introduced to remap inputs into standard distribution[11]. To better handle pruning on the Batch-Normalization layer, [12] proposed a channel-wise scaling factor in generating sparsity matrix and compacting model. However, for the purpose of compressing YOLO-v3 to a reasonable size and fitting in the Jetson Nano which supports only 472 GFLOPs (Gigabytes Floating Point Operation per second) of GPU. We can choose to obtain a compressed YOLO Network through structured pruning other than the unstructured method (which usually involves neuron level or filter level pruning in granularity). On the other hand, the unstructured approach is more demanding in computation resource on hardware implementation. Structured Pruning parsimoniously helps reduce redundant space and testing time and exhibit relatively good performance in the following training procedures defined as below and can be referred in Fig.5:

1) We set a sensitivity ratio: $\gamma$, (i.e., channel scaling factor) to eliminate unnecessary weights in a layer, and rank all the weights based on significance, i.e., saliency based.
2) Prune all the Batch Normalization (BN) layer first as implied by Fig. 6 and discard all the weights that are below $\gamma$, generating sparsity matrices
3) Resizing the Convolutional layer in YOLO-v3 Network based on the former and latter BN layers.
4) Conduct a saliency-based pruning on a fully connected layer by constructing a zero-masking matrix as indicated in Fig. 7.

Hence, we can illustrate our pruning scheme over the Batch Normalization layer. Let $\beta = \{x_1, x_2, ..., x_m\}$ where $\beta$ denotes a mini-batch, and assume $y$ is the output array of this layer, thus we have:

$$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\} \quad (8)$$

where $\gamma, \beta$ are 2 trainable parameters in pruning. We define $\mu_\beta$, $\sigma^2_\beta$ to be the mean and variance of mini-batch $\beta$ respectively. Therefore, we can construct deviation of $x_i$ in the following form:

$$\hat{x} = (x_i - \mu_\beta) (\sigma^2_\beta + \epsilon)^{-\frac{1}{2}} \quad (9)$$

$$y_i = \gamma \hat{x}_i + \beta \quad (10)$$

Hence, we can further compute the differential equation based on our loss function $L$ indicated as follow:

$$L = \sum_{(x,y)} l(f(x, \omega), y) + \lambda \sum_{y \in \Gamma} g(y) \quad (11)$$

Thus, the differential solution can be concluded in the below equations:

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial y_i} \cdot \gamma \quad (12)$$

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \quad (13)$$

$$\frac{\partial l}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \cdot \hat{x}_i \quad (14)$$
\[ \frac{\partial l}{\mu_{\beta}} = \sum_{i=1}^{m} \frac{\partial l}{\partial x_i} \left( \sigma_{\beta}^2 + \epsilon \right)^{-\frac{1}{2}} \tag{15} \]

\[ \frac{\partial l}{\partial \sigma_{\beta}} = -\frac{1}{2} \sum_{i=1}^{m} \frac{\partial l}{\partial x_i} \cdot \left( x_i - \mu_{\beta} \right) \cdot \left( \sigma_{\beta}^2 + \epsilon \right)^{-\frac{3}{2}} \tag{16} \]

\[ \frac{\partial l}{x_i} = \frac{\partial l}{\partial x_i} \cdot \left( \sigma_{\beta}^2 + \epsilon \right)^{-\frac{1}{2}} + \frac{\partial l}{\partial \sigma_{\beta}} \cdot \frac{2(x_i - \mu_{\beta})}{m} + \frac{\partial l}{\mu_{\beta}} \cdot \frac{1}{m} \tag{17} \]

Empirically, we could set lambda in a relatively small range between $10^{-5}$ to $10^{-4}$.

---

**FIGURE 5.** Pruning from initial Network to compact Network

**FIGURE 6.** Pruning Process over Batch-Normalization Layers

**FIGURE 7.** Pruning fully connected layer with masks

**TABLE 3.** Object Detection speed and accuracy on COCO dataset of Pruned models and its counterpart model, Latency is measured by Frames Per Second (FPS) with different core settings on Jetson NANO.

| Training Type | Latency/FPS (256 core) | Latency/FPS (128 core) |
|---------------|------------------------|------------------------|
| Unpruned      | 52.282                 | 56.662                 |
| Pruned        | 50.720                 | 55.432                 |
3.3. IMPLEMENTATION AND EVALUATION OF PRUNING

We implemented the pruned YOLO-v3 architecture with its counterpart model and benchmarked it with various core settings on Jetson NANO and comparing the performance on the COCO dataset [13]. As can be seen in Table 3, our pruning experiment improved latency by more than 1 FPS on multi-cores Jetson NANO.

Notably, we also obtained an accuracy improvement since our pruned YOLO Network overcomes the overfitting problems from the regularization of structured pruning shown in Fig. 8 and Fig. 9.

In Table 4, we utilized Pytorch Open-Source API to train and employ our model into Jetson NANO CPU, and we also see an enhanced inference time: 10s faster and accuracy by 0.08% for the real-time Object Detection task on the COCO dataset.

| Training Type | Accuracy (avg.) | Latency (s) |
|---------------|-----------------|-------------|
| Unpruned      | 82.57           | 998.976     |
| Pruned        | 82.49           | 985.114     |

4. CONCLUSION

In this paper, we explored two major methods: quantization and pruning in model compression and delivered solid and verifiable improvements in quantization trade-offs between accuracy and speed by converting AlexNet into INT8 form.

We also advanced the YOLO-v3 architecture by pruning and deployed it with lower memory footprints. In the real-time Object Detection task, we also improved the performance through a regularization pruning scheme. However, this paper should also adopt more hardware platforms for deployment experiments for comparison study since different back-end and accelerated hardware may result in different performances. In the near future, possible model compression directions such as knowledge distillation, low-rank factorization should be involved in experiments. Challenges also come from the rethinking of our current methods as stated in [17] and ideas that automatically learning from one-shot pruning can not compete in accuracy with non-iterative pruning as shown in [19].

Acknowledgments

In addition, Hao and Xiao would like to thank our team member, Shuyu Yang, for his early data and literature collection.
References
[1] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Advances in neural information processing systems, 2012, 25: 1097-1105.
[2] Cheng Y, Wang D, Zhou P, et al. A survey of model compression and acceleration for deep neural networks[J]. 2017, arXiv preprint arXiv:1710.09282.
[3] Zhang X, LeCun Y. Text understanding from scratch[J]. arXiv preprint arXiv:1502.01710, 2015.
[4] Jacob B, Kligys S, Chen B, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference[C]. Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 2704-2713.
[5] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In Proceedings of the 32nd International Conference on Machine Learning (ICML-15), pages 1737–1746, 2015. 2
[6] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[J]. arXiv preprint arXiv:2005.14165, 2020.
[7] Lagler K, Schindelegger M, Böhm J, et al. GPT2: Empirical slant delay model for radio space geodetic techniques[J]. Geophysical research letters, 2013, 40(6): 1069-1073.
[8] Google. Tensorflow Lite. https://www.tensorflow.org/mobile/tflite. 2, 3, 4, 11
[9] S. Kligys, S. Sivakumar, et al. TensorFlow quantized training support. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/quantize. 5, 6
[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, abs/1510.00149, 2, 2015. 1
[11] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML’15, pages 448–456. JMLR.org, 2015. 5, 6
[12] Liu, Zhuang, et al. "Learning efficient convolutional networks through network slimming." Proceedings of the IEEE international conference on computer vision. 2017.
[13] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Doll’ar, and C. L. Zitnick. Microsoft coco: Common objects in context. In European conference on computer vision, pages 740–755. Springer, 2014. 2
[14] Redmon J, Farhadi A. Yolov3: An incremental improvement[J]. arXiv preprint arXiv:1804.02767, 2018.
[15] Li H, Liu H, Ji X, et al. Cifar10-dvs: an event-stream dataset for object classification[J]. Frontiers in neuroscience, 2017, 11: 309.
[16] Lin M, Chen Q, Yan S. Network in network[J]. arXiv preprint arXiv:1312.4400, 2013.
[17] Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." arXiv preprint arXiv:1803.03635 (2018).
[18] Cass S. Nvidia makes it easy to embed AI: The Jetson nano packs a lot of machine-learning power into DIY projects-[Hands on][J]. IEEE Spectrum, 2020, 57(7): 14-16.
[19] Zhu M, Gupta S. To prune, or not to prune: exploring the efficacy of pruning for model compression[J]. arXiv preprint arXiv:1710.01878, 2017.
[20] Redmon J, Joseph A, Farhadi A. YOLO-v3, 2018, https://pjreddie.com/darknet/yolo/