OPTIMIZING THE PERFORMANCE OF CONVOLUTIONAL NEURAL NETWORKS ON RASPBERRY PI FOR REAL-TIME OBJECT DETECTION

Hyun Woo Jung

Hankuk Academy of Foreign Studies, Seoul, Republic of Korea

ABSTRACT

Deep learning has facilitated major advancements in various fields including image detection. This paper is an exploratory study on improving the performance of Convolutional Neural Network (CNN) models in environments with limited computing resources, such as the Raspberry Pi. A pretrained state-of-art algorithm for doing near-real time object detection in videos, YOLO ("You-Only-Look-Once") CNN model, was selected for evaluating strategies for optimizing the runtime performance. Various performance analysis tools provided by the Linux kernel were used to measure CPU time and memory footprint. Our results show that loop parallelization, static compilation of weights, and flattening of convolution layers reduce the total runtime by 85% and reduce memory footprint by 53% on a Raspberry Pi 3 device. These findings suggest that the methodological improvements proposed in this work can reduce the computational overload of running CNN models on devices with limited computing resources.

KEYWORDS

Deep Learning, Convolutional Neural Networks, Raspberry Pi, real-time object detection

1. INTRODUCTION

Deep learning has fundamentally transformed the way we think about machine learning and artificial intelligence, and numerous research and real-world application of deep learning in various fields such as medicine [1], finance [2], and image analysis [3]. Recent trends in deep learning focuses heavily on improving the accuracy and reducing the training time of the model through various methods. However, the biggest impediment in bringing deep learning to everyday usage is mainly the amount of computation that needs to be done in a large cluster or GPU setting – one that is not available in everyday devices that most people interact with, such as mobile phones or embedded devices.

Convolutional neural networks (CNN), one of the most successful deep learning techniques, are inspired by the visual cortex of the brain and are designed to process multiple types of data. This paper studies ways to enhance the runtime (feedforward) performance of CNN models through profile-guided optimizations and compile optimizations on an embedded device. To bring in a realistic context to the research, this paper focused on optimizing the feedforward runtime of a pre-trained CNN-based model used for real-time video processing called YOLO ("You-Only-Look-Once") model [3] to perform better on a Raspberry Pi 3 device, which runs a 64-bit ARMv8 CPU with a 1GB DDR2 RAM. The rationale behind

Natarajan Meghanathan et al. (Eds) : ACITY, AIAA, DPPR, CNDC, WIMNET, WEST, ICSS - 2019
pp. 103-110, 2019. © CS & IT-CSCP 2019 DOI: 10.5121/csit.2019.91711

studying ways to optimize performance of a pre-trained model is that most training happens on a server-class machines with access to plenty of resources. However, the context at which these models need to perform at runtime is usually equipped with much less resource than the context at which they were trained. Therefore, the focus for this study was set at optimizing a model that was already trained, instead of trying to create an optimized or compressed CNN model at train-time.

2. FORMAT GUIDE

2.1. Background

2.1.1. You-Only-Look-Once (YOLO) Algorithm

YOLO is an object detection algorithm used for real-time object detection [3]. YOLO is a state-of-art object detection algorithm that outperforms most known peers by only applying a single neural network to the entire image instead of reapplying classifiers at multiple locations of the image, which is what most of the latest object detection algorithms use. Because of this, YOLO has a significant performance advantage compared to its peers which is that it only needs to perform a single network evaluation forits task, as opposed to hundreds or thousands of evaluations that need to be done on its peers such as Faster R-CNN [4].

2.1.2. Previous Studies on Optimizing Performance of Convolutional Neural Network (CNN) models

Many researchers have already attempted to optimize the performance of CNN models using various methods[4]. Gong, Yunchao, et al. suggested using singular vector decomposition (SVD) to compress each layer in the network. In particular, this method compresses each weight matrix by only keeping the singular values with largest magnitude, which significantly reduces the memory footprint of the model at feedforward runtime while keeping the accuracy loss to roughly 1%. [5]
In CNN, the "flattening" step involves coversion of multidimensional convolution matrices to 1-dimensional array. For instance, a convolution matrix of size X x Y x Z becomes 3 different convolution matrices, each with size X x 1 x 1, Y x 1x 1, and Z x 1 x1, respectively). has been done to reduce memory – this reduces memory footprint significantly because the number of parameter gets reduced from O(XYZ) to O(X + Y+ Z). Jin, et al. showed that this method can reduce both training and classification time by more than half. [6]

Lastly, Han, et al. proposed a way of reducing network size by iterative pruning of the network. [8] In a method similar to pruning a decision tree, Han proposed a way of removing unimportant nodes out of the network by pruning the network for connections with weights whose magnitude is less than a certain threshold. Following this, the network gets retrained with a new structure. The retrained network then goes through another pruning process in an iterative manner.

2.2. Measuring Performance of CNN Model

Feedforward runtime speed is the most important factor to consider when measuring the performance of deep neural networks. Many efforts have been undertaken to predict the runtime of a CNN model at train time. [7] The most straightforward metric for evaluating the runtime of a model is FLOPS, which is the number of floating point operations that need

to be performed. This is due to the nature that floating point operations are very expensive, and much of the time during runtime is spent on performing floating point operations.

However, Yunchao, et al. showed that the relationship between a CNN model's FLOPS and its runtime is nonlinear, and may vary by 3-4 times depending on many different conditions. [5] Several factors were pointed as culprit for such nonlinearity, including cache optimizations, disk I/O, as well as memory footprint.

Memory is another important factor to consider when measuring the performance of a CNN model due to the fact that memory is often just much of a constraining factor in an embedded environment as CPU is – perhaps even further. For example, a Raspberry Pi 3 has a memory size of 1GB, whereas most server machines used for deep learning run with 64GB or more memory – a difference of 64 times. Excessive memory usage may lead to disk swapping, which impacts the runtime of the model very heavily because disk swapping may involve disk I/Os. Large memory footprint negatively impacts cache performance as well.

2.3. Method

To optimize the performance of YOLO model on a Raspberry Pi, an architecture-specific profile guided approach was taken to identify potential performance bottlenecks at feedforward runtime. The profile data was collected by using the perf tool on Raspbian OS, on a Raspberry Pi 3 device by running pre-trained YOLO models on DarkNet [], a small open-source neural network framework. Various model sizes were used to ensure that the profile data was neutral.

2.4. Results

2.4.1. CPU Profile

To find out which part of the DarkNet contributes most heavily to the runtime of classification task, a CPU based-sample was taken on a Raspberry Pi 3 using perf [4]. perf is a performance analysis tool available on Linux systems and it reports the percentage of CPU time spent in specific parts of the code. perf was run on a DarkNet process that was running a pre-trained YOLO model which was a 106-layer CNN network.

```
Samples: 163K of event 'cycles:ppp', Event count (approx.): 167851257924
Overhead  Command  Shared Object      Symbol
  87.86%  darknet  darknet            [.] gemm_nn
   2.26%  darknet  libm-2.23.so       [.] __cos_avx
   2.00%  darknet  libm-2.23.so       [.] __ieee754_log_avx
   1.80%  darknet  libm-2.23.so       [.] __sin_avx
   0.77%  darknet  darknet            [.] rand_normal
   0.73%  darknet  darknet            [.] im2col_cpu
   0.68%  darknet  darknet            [.] im2col_get_pixel
   0.66%  darknet  darknet            [.] activate
   0.26%  darknet  libc-2.23.so       [.] __random
   0.25%  darknet  darknet            [.] activate_array
   0.24%  darknet  darknet            [.] normalize_cpu
   0.21%  darknet  libc-2.23.so       [.] __random_r
   0.21%  darknet  darknet            [.] fill_cpu
   0.18%  darknet  darknet            [.] shortcut_cpu
   0.18%  darknet  darknet            [.] copy_cpu
   0.15%  darknet  darknet            [.] make_convolutional_layer
   0.11%  darknet  [kernel.kallsyms]  [k] clear_page_erms
   0.08%  darknet  libc-2.23.so       [.] rand
   0.08%  darknet  [kernel.kallsyms]  [k] native_irq_return_iret
   0.07%  darknet  [kernel.kallsyms]  [k] error_entry
   0.07%  darknet  darknet            [.] add_bias
   0.06%  darknet  darknet            [.] gemm_cpu
   0.06%  darknet  [kernel.kallsyms]  [k] sync_regs
   0.06%  darknet  darknet            [.] scale_bias
   0.06%  darknet  [kernel.kallsyms]  [k] swapgs_restore_regs_and_return_to_usermode
   0.06%  darknet  [kernel.kallsyms]  [k] copy_user_enhanced_fast_string
   0.05%  darknet  libm-2.23.so       [.] __ieee754_exp_avx
   0.05%  darknet  [kernel.kallsyms]  [k] get_page_from_freelist
```

Figure 1. CPU profile

2.4.2. Memory Profile

To find out how much memory is being used by DarkNet at runtime, valgrid [9] was used to collect memory profile on DarkNet. valgrind is a Linux-based tool that is used for various heap analysis purposes, such as memory profiling and memory leak detection. Similar to what perf does with CPU, it samples the heap regularly to construct a map of the heap usage of a process over time.

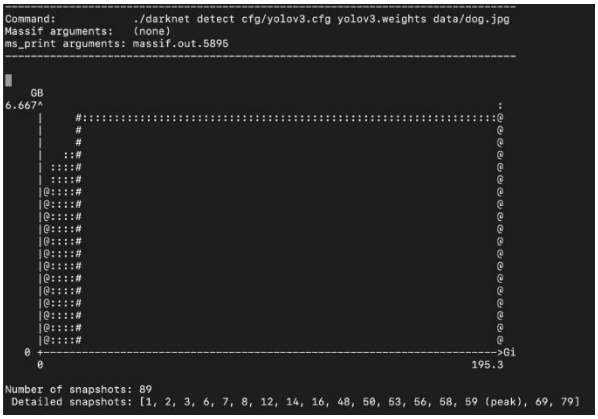

Figure 2. Memory profile data collected on DarkNet valgrind was run

Figure 2. Memory profile data collected on DarkNet valgrind was run on the same hardware, running the same network as part 1). Figure 2 shows that roughly 6.6GB of memory footprint is hit very quickly and stays at 6.6GB until classification is complete. Because this exceeds the amount of memory available on a Raspberry Pi 3, or most embedded device, it clearly shows that the computational overload of the model needs to be reduced.

2.4.3. Loop Parallelization

In Section III, CPU profile data collected using perf clearly pointed at over 87% of the CPU time was spent inside a single function, gemm_nn, in DarkNet. gemm_nn is a function that is used to multiply two matrices containing the weights – because weights are represented as floating point matrices, this ends up performing a lot of floating point operations.

```
void gemm_nn(int M, int N, int K, float ALPHA,
        float *A, int lda,
        float *B, int ldb,
        float *C, int ldc)
{
    int i,j,k;
    for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            register float A_PART = ALPHA*A[i*lda+k];
            for(j = 0; j < N; ++j){
                C[i*ldc+j] += A_PART*B[k*ldb+j];
            }
        }
    }
}
```

Figure 3. Source code of DarkNet - gemm_nn

As shown in Figure 3, gemm_nn updates a 3 dimensional matrix, and is a clear example of a “embarrassingly parallel” workload. To speed this up, a loop parallelization was performed

using OpenMP [10]. Using OpenMP, another binary was compiled that made use of the loop parallelization, and the CPU time was evaluated using the same YOLO model.

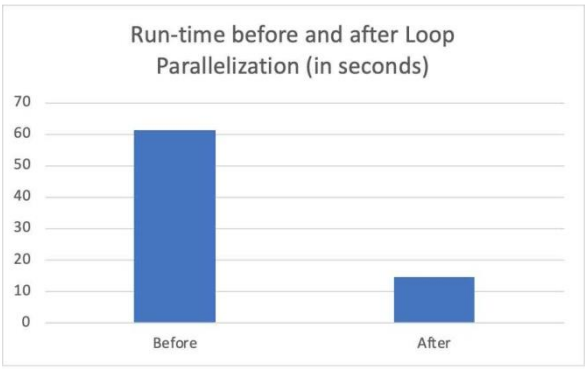

Figure 4. Runtime before and after loop parallelization

Figure 4 shows the result of the overall runtime before and after the loop parallelization. The overall runtime reduced from 61 seconds to 13 seconds, which is an overall improvement of over four times.

2.4.4. Static Compilation of Weights

Various methods were explored to reduce the total memory footprint of DarkNet during runtime, and one of them was to turn the model into a statically compiled single executable. More specifically, the weight vectors were declared as static constant variables in the C code, and the entire model was re-compiled together with DarkNet into a single executable.

Doing this reduces the memory footprint because it eliminates the need of dynamically-allocated memory which may reduce in memory fragmentation in the kernel heap. In addition to saving memory footprint from reduced memory fragmentation, it may also reduce the total CPU time as the extra cost of system calls to dynamically allocate heap space is removed.

To statically compile the weight with the model, a simple Python script was written to take in a pre-trained weight file and generate a C header file that contains the weight matrix as hard-coded static variables, which can be then included in the DarkNet CMakeFile to be linked together into a single executable.

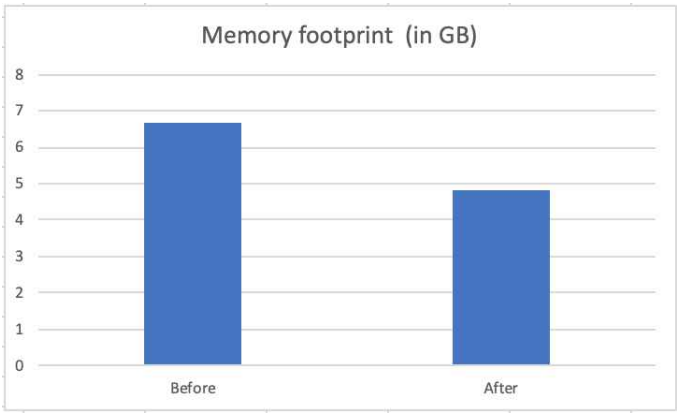

Figure 5. Memory footprint before and after static compilation of weight matrices

As shown in Figure 5, static compilation of weight matrices yielded roughly 28% reduction in the memory footprint of the process. While it did not yield as big of a reduction in memory footprint as loop parallelization did for CPU time, it successfully reduced a significant portion of the memory footprint.

Doing this means that to use different model, the entire network needs to be recompiled and this reduces the portability of models because they need to be compiled on the target device again. Figure 6 is a visualization of what happens to the overall input and output architecture of the classification system as a result of static compilation. As shown in the figure, weight vector can not be passed as a parameter to DarkNet because the network has already been hard-coded together with DarkNet. This sacrifices the flexibility of the network.

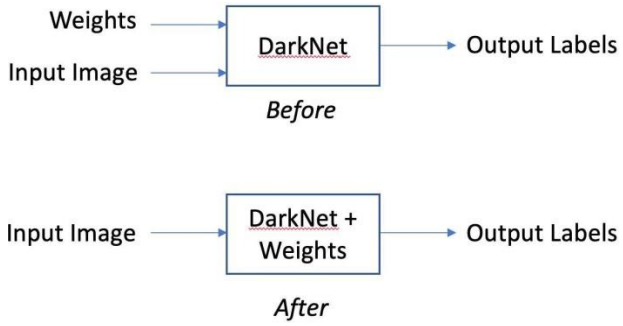

Figure 6. Visualization of the Model Change Before and After Static Compilation of Weights

However, in most embedded environments, this is a reasonable tradeoff because embedded devices would run a pre-trained model and reuse the same model until further update from the server with a newly-trained model anyway. Therefore, it can be assumed that the introduction of additional compilation time once per each set of newly trained weights is a reasonable additional cost for reducing memory footprint of the model.

2.4.5. Flattening Weights

Previous studies have shown that 3D convolutions improve the feedforward performance of CNN models (Ref for Jin et al. paper). More specifically, 3D convolutions dramatically improve the feedforward runtime of the model by breaking apart the convolutional layers in the network from a 3-Dimensional convolution matrix into a series of 1-Dimensional convolution vectors. This reduces the total number of parameters in the network by a large amount. The reduction in the total number of parameters can reduce both the feedforward time as well as the memory footprint of the model.

Using this method, a flattening of the weight vector was performed after the static compilation weights has been done. Weight vectors for the convolutional layers were divided into three separate sequences of 1 dimensional convolutional layers and re-measured for performance using perf and valgrid utilities in Linux.

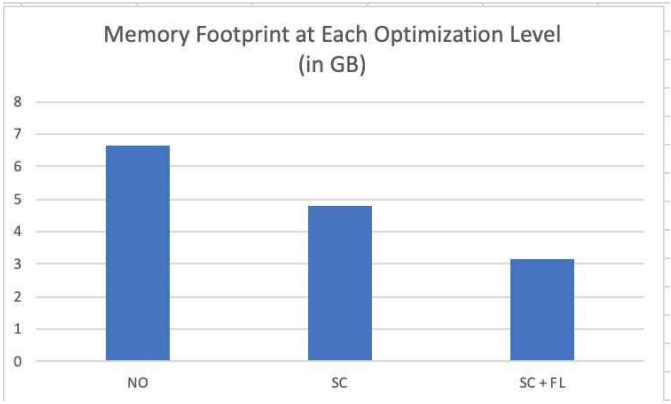

Figure 7. Memory Footprint at Each Optimization Level

Figure 7 shows that use of both static compilation and flattening (SC+FL) resulted in a total memory footprint of about 3.1 GB, which was a 53% improvement compared to the model with no optimization applied (NO). Compared to the model with just static compilation optimization applied, the total memory footprint was reduced by 36%.

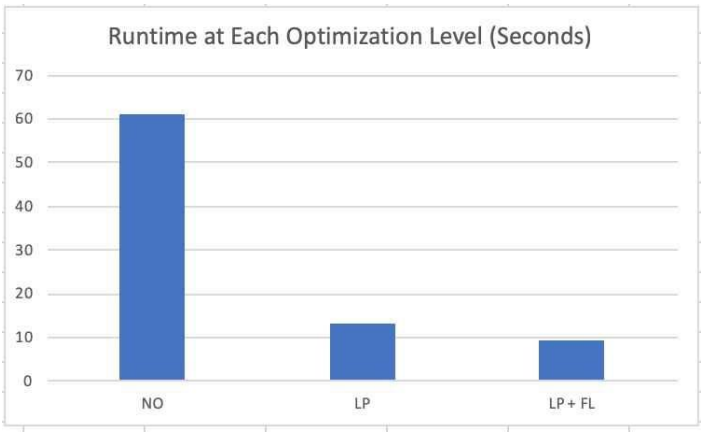

Figure 8. Feedforward Runtime at Each Optimization Level

Figure 8 shows the feedforward runtime of the same model at each optimization level. When loop parallelization and flattening is both applied (LP + FL), the runtime was reduced by 32% compared to the model where just loop parallelization was applied (LP). Compared to the original model without any optimization, the feedforward runtime was reduced by 85%..

3. CONCLUSIONS

In this study various methods were explored to improve the performance of YOLO model in an embedded setting. Compared to similar studies, this study took a real-life example and showed that various procedures can be implemented to improve performance in a limited context.A total of three different methods including loop parallelization, static compilation of weight matrices, and flattening weight matrices were applied together to improve the overall feedforward time by more than 85% and the memory footprint by 45%, without a significant reduction in the accuracy of the model.

Since this study was a largely exploratory study, there are some limitations. The current work was focused on a single type of CNN model. Additionally, due to limitation in computing

power, only a limited number of models were trained. Future work will focus on scaling the methodology to more generic models. The model optimization procedure can also be automated by creating an architecture-agonistic framework that allows any arbitrary model to be accelerated via automatic loop parallelization through static analysis, dynamic generation of statically compiled weight matrices, as well as flattening of the convolution layers.

REFERENCES

[1] Miotto R, Wang F, Wang S, Jiang X, Dudley JT. (2018). Deep learning for healthcare: review, opportunities and challenges. Brief Bioinform. Nov 27;19(6):1236-1246

[2] Heaton, J. B., Polson, N. G., and Witte, J. H. (2017) Deep learning for finance: deep portfolios. Appl. Stochastic Models Bus. Ind., 33: 3– 12.

[3] Joseph Redmon, Ali Farhadi. (2018). YOLO v3: An Incremental Improvement. ArXiv.

[4] S. Ren, K. He, R. Girshick and J. Sun. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, 1 June.

[5] Yanchao Gong, Liu Liu, Ming Yang, Lubomir Bourdev. (2015). Compressing Deep Convolutional Networks Using Vector Quantization. International Conference on Learning Representations.

[6] Jonghoon Jin, Aysegul Dundar, Eugenio Culuciello. (2015). Flattened Convolutional Neural Networks for Feedforward Acceleration. International Conference on Learning Representations.

[7] Cheng, Yu, Wang, Zhou, Zhang, & Tao. (2018). Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. Retrieved from https://www.gwern.net/docs/ai/2018-cheng.pdf

[8] Song Han, Jeff Pool, John Tran, William J. Dally. (2015). Learning both Weights and Connections for Efficient Neural Network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama and R. Garnett (Eds). Advances in Neural Information Processing Systems 28. Red Hook, NY: Curran Associates, Inc.

[9] Valgrind. (Visited 2019). http://valgrind.org.

[10] OpenMP. (Visited 2019). API Specification for Parallel Programming. https://www.openmp.org.

Authors

Hyun Woo Jung
Curr. Hankuk Academy of Foreign Studies.
Research Interests: Computer Science, Data, Machine Learning