APSIPA Transactions on Signal and Information Processing, 2023, 12, e3

Original Paper

Convolutional Neural Networks Inference Memory Optimization with Receptive Field-Based Input Tiling

Weihao Zhuang*, Tristan Hascoet, Xunquan Chen, Ryoichi Takashima, Tetsuya Takiguchi and Yasuo Ariki

Kobe University, Kobe 657-8501, Japan

ABSTRACT

Currently, deep learning plays an indispensable role in many fields, including computer vision, natural language processing, and speech recognition. Convolutional Neural Networks (CNNs) have demonstrated excellent performance in computer vision tasks thanks to their powerful feature-extraction capability. However, as the larger models have shown higher accuracy, recent developments have led to state-of-the-art CNN models with increasing resource consumption. This paper investigates a conceptual approach to reduce the memory consumption of CNN inference. Our method consists of processing the input image in a sequence of carefully designed tiles within the lower subnetwork of the CNN, so as to minimize its peak memory consumption, while keeping the end-to-end computation unchanged. This method introduces a trade-off between memory consumption and computations, which is particularly suitable for high-resolution inputs. Our experimental results show that MobileNetV2 memory consumption can be reduced by up to 5.3 times with our proposed method. For ResNet50, one of the most commonly used CNN models in computer vision tasks, memory can be optimized by up to 2.3 times.

Keywords: Convolutional neural network, memory optimization, receptive field.

*Corresponding author: Weihao Zhuang, zhuangweihao@stu.kobe-u.ac.jp.

Received 20 March 2022; Revised 16 October 2022
ISSN 2048-7703; DOI 10.1561/116.00000015

1 Introduction

Recent developments in computer vision have led to the introduction of new models with improved accuracy. Unfortunately, higher accuracy often comes at the cost of increased resource consumption. Given their high resource consumption, state-of-the-art models are difficult to deploy in practical edge device use cases, in which resources are limited. Although it is possible to delegate the inference from edge devices to remote servers, this solution comes with additional network resource consumption, latency, and additional data privacy risks. Hence, optimizing the computing resources needed by CNNs to run on edge devices without affecting the accuracy is a challenge of practical importance.

In this paper, we focus on one particular aspect related to the resource consumption of CNNs: we aim to optimize the memory consumption of CNNs during the inference phases. The memory used by a CNN model consists of two parts – activations and parameters. The number of parameters a given CNN has is fixed when the model is designed. The amount of device memory used by activations is largely related to the resolution of the input. Figure 1 shows the ratio of the memory consumption caused by the activations to the memory consumption due to parameters of the model for different resolutions; i.e., the memory consumption of the activations divided by the memory consumption of the parameters. As the input image resolution increases, the memory

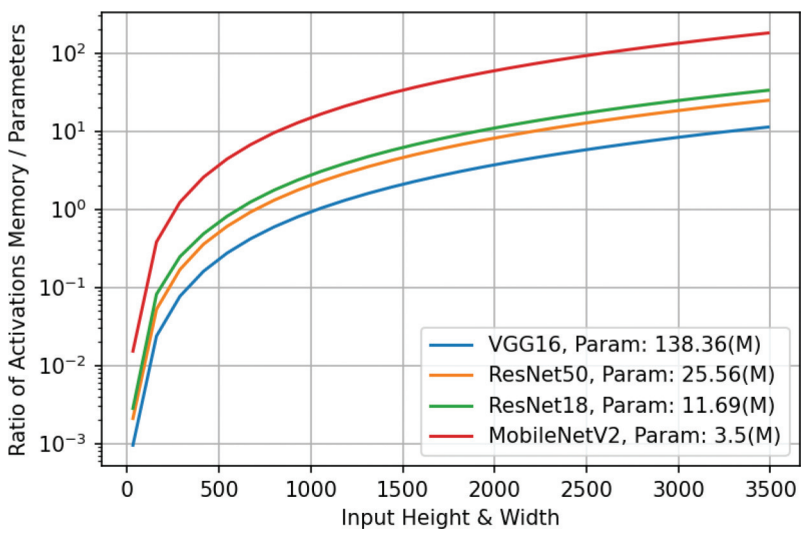


Figure 1: The ratio of activations memory consumption to the parameters at different resolutions in CNN inference.

bottleneck in inference is gradually dominated by activations. For models with a smaller number of parameters, the effect of activations on memory consumption will become more pronounced as the resolution increases. For MobileNetV2 [16], the model with the smallest number of parameters among these models, the memory consumption for activations is 16 times higher than the parameters when the input resolution is 1024×1024.

Many computer vision benchmarks and implementations use image resolutions of 224×224 pixels, which is a common standard nowadays. This relatively low resolution explains why activations have not been considered a bottleneck in inference memory consumption, and most works aiming to compress networks for inference have focused on reducing the footprint of the model parameters. However, using higher resolution input is important for some practical computer vision tasks: Sabottke *et al.* [15] proposed that a CNN network trained with a high resolution has better performance when dealing with medical images. In industrial defect detection applications, the resolution of the image is often relatively high [17, 18]. Therefore, it would be beneficial to reduce the memory bottleneck caused by activations in the inference phase.

A straightforward method to reduce the memory consumption of CNN inference consists of sequentially processing input images in spatially arranged tiles, as illustrated in Figure 4a. That way, the computations of the model are equivalent to sequentially processing low-resolution images, hence reducing the inference memory cost. However, naively processing images in spatial tiles will result in approximate computations due to artifacts arising at the edges of each tile, which affects the accuracy of the CNN. This is because pixels at the edge of each tile lose the information of their neighbor pixel because they are typically replaced by zero-padding in most practical implementations. This causes information at the edge to be lost. These artifacts can be mitigated, and even completely removed, using overlapping tiles to guarantee a computation exactly similar to the standard model execution. However, overlapping tiles introduce a computational cost overhead due to redundant computations at the edge of the tiles. This highlights a fundamental trade-off between memory consumption and computations, which is the topic of this paper.

We propose using the receptive fields of neurons at arbitrary layers to parameterize tiles that ensure that the output is unchanged and without unnecessary computation. Given these tiles, we are able to quantify the amount of redundant computation, and, hence, the memory overhead of a given choice of tiles. Similarly, we are able to compute the memory reduction achieved for different tiles. This provides us with formulas for calculating the trade-off between memory consumption and computational overhead later for different models and choices of tiling.

Several methods have been proposed to reduce the memory consumption associated with inference. These can be broadly divided into three categories:

model compression [4–6, 10], light-weight model design [2, 7–9, 12, 16, 21], and low-level operation implementation optimization [3, 19]. Most of these methods are carried out from the perspective of optimizing the memory consumption of the parameters.

Our method is based on using the receptive field to tile the input image, and then optimizing the memory bottleneck caused by activations during CNN inference. The method of calculating an arbitrary neural network receptive field proposed by Araujo *et al.* [1] provides us with tools for theoretical analysis. The method most related to our approach is proposed by Wu *et al.* [20], which also tiles the input to several parts and feeds them into the CNN model. However, this method does not use receptive field-based tiling, which makes it impossible to study the exact trade-off we propose in this paper.

The remainder of this paper is organized as follows: We start by presenting prerequisite knowledge together with our methodology in Section 2. In Section 3 we use a toy parametrization of CNN architecture to derive the exact trade-off between memory consumption and computational overhead. Finally, in Section 4 we analyze the trade-off between computational overhead and memory consumption for some standard CNN architectures using our proposed method.

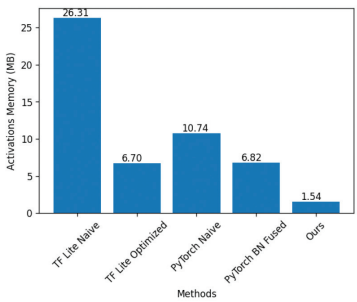
2 The Proposed Method

Section 2.1 presents background information about the memory consumption of CNN models in the inference phase. The basics of the receptive field will be introduced in Section 2.2. Section 2.3 combines these concepts to formalize our proposed method.

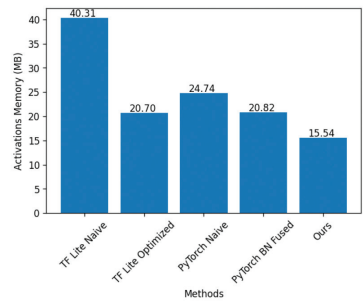
2.1 Memory Consumption of CNN Inference

When a CNN performs inference, the total memory consumption is the sum of the parameters and activations stored in the device's memory. A simple memory allocation strategy is to allocate memory buffers for all the activations and parameters needed by the model. The Tensorflow lite [11] naive method uses exactly this strategy. However, the memory cost of this strategy is very high. PyTorch [13] allocates output buffers before and deallocates input buffers after the execution of each layer. This strategy greatly reduces memory usage at the cost of overhead for dynamic memory management. The Tensorflow Lite optimized method uses the method proposed by Pisarchyk *et al.* [14] to optimize memory consumption during inference. In [14], to reduce the overhead of dynamic memory management, the authors allocate memory buffers that can be reused during the execution of the model. In addition, [14] greedily allocates the smallest possible memory buffers when the model is executed.

Figure 2 illustrates the memory consumption of MobileNetV2 using different inference strategies. Figure 2a shows only the memory occupied by the activations, while Figure 2b shows the memory consumption including the activations and parameters. We can see that the memory consumption after Tensorflow Lite optimization is very close to that of "PyTorch Naive", although there is still a difference of about 4 MB.



(a) The amount of memory consumed by the activations.



(b) The amount of memory consumed by activations and parameters.

Figure 2: Comparison of different methods for MobileNetV2 memory optimization (224×224 Input).

Tensorflow Lite, as a successful deployment framework, usually fuses Batch Normalization (BN) into convolution to reduce the inference latency, while PyTorch performs all operators as separate computational nodes by default. Our approach is optimized base on PyTorch Naive, which we describe in Section 2.3. Although we could choose to optimize the PyTorch implementation after BN fusion, we do not do that for the sake of comparison with the standard PyTorch implementation.

Our proposed method can optimize 4.35 times the activation consumption compared to Tensorflow Lite optimized method. Although our approach has a huge memory optimization for activations, it comes at the cost of increased computation. Moreover, for the optimization method with tiling of 16, we need to execute the model 16 times, which results in a certain degree of memory management time consumption and low-level implementation time consumption.

Table 1 and Figure 3 show the memory footprint of a simple CNN model at inference. Table 1 shows the architecture of this CNN model. This model consists of six convolution layers and two max-pooling layers. Every two convolution layers are followed by a max-pooling layer, which downsamples the activations by a factor of 2 in both spatial dimensions.

Figure 3 shows the memory consumption footprint of the model during the inference phase. We drew Figure 3 based on the PyTorch memory management

Table 1: Toy network architecture.

Layer Name	Output Size (Channel, Height, Width)	Configuration (Kernel, Channel, Stride)
Input	$3 \times 32 \times 32$	-
Conv1, Conv2	$32 \times 32 \times 32$	3×3, 32, 1
Maxpool1	$32 \times 16 \times 16$	3×3, 2
Conv3, Conv4	$64 \times 16 \times 16$	3×3, 64, 1
Maxpool2	$64 \times 8 \times 8$	3×3, 2
Conv5, Conv6	$128 \times 8 \times 8$	3×3, 128, 1

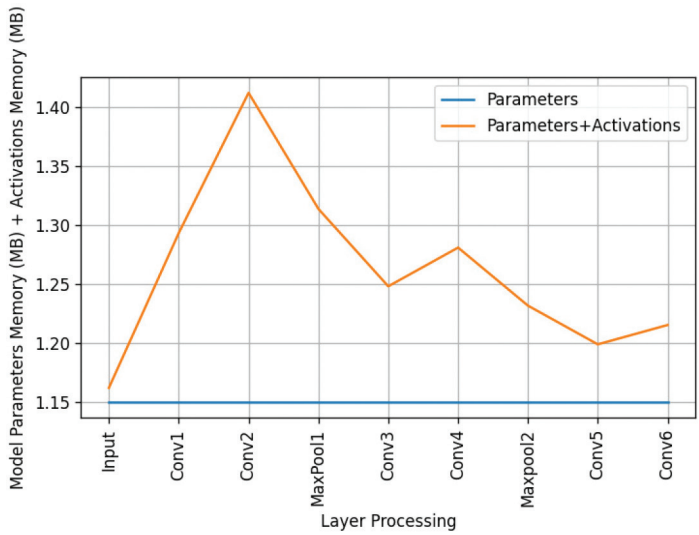


Figure 3: Memory consumption footprint of toy CNN model inference (FP32).

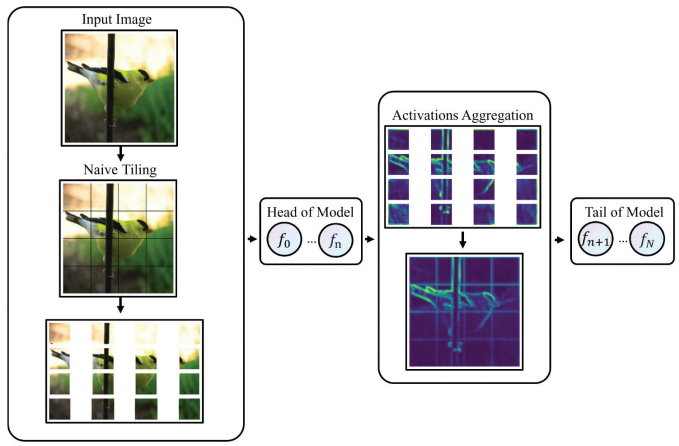
approach as a motivation to introduce our memory optimization approach. The blue line represents the amount of device memory occupied by the model's parameters. The parameters are always stored in the device memory. The orange curve shows the number of activations and the amount of memory occupied by the device by the parameters. Memory consumption peaks at some stage of the CNN model inference, such as Conv2 in Figure 2. Our goal is to reduce this peak.

2.2 The Receptive field of CNN models

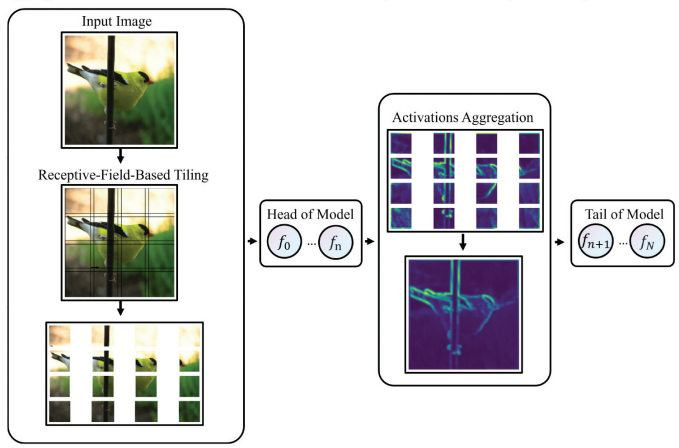
This section introduces the basics of the receptive field related to this paper using a two-layer convolutional model and a CNN model with a residual block.

The concepts and information covered in this paper have been introduced by [1].

The receptive field (RF) is defined as the size of the input region that produces the feature. Figure 5a illustrates the effect of the number of layers on the RF size. The blue trapezoid on the left illustrates the case where the size of the RF is 5 after two layers of 3×3 convolution. That is, there are five elements in the one-dimensional input that contribute to the activations of the output. The light orange trapezoid on the right illustrates the case where the



(a) Naive Method: The tiles obtained after naive tiling are fed to the head of the CNN model, and the activations obtained from each tile are aggregated and fed to the tail of the CNN model to complete the remaining inference process.



(b) Proposed Method: The tiles obtained after tiling based on the receptive field are fed to the head of the CNN model, and the activations obtained from each tile are aggregated, and fed to the tail of the CNN model to complete the remaining inference process.

Figure 4: Naive method and proposed method.

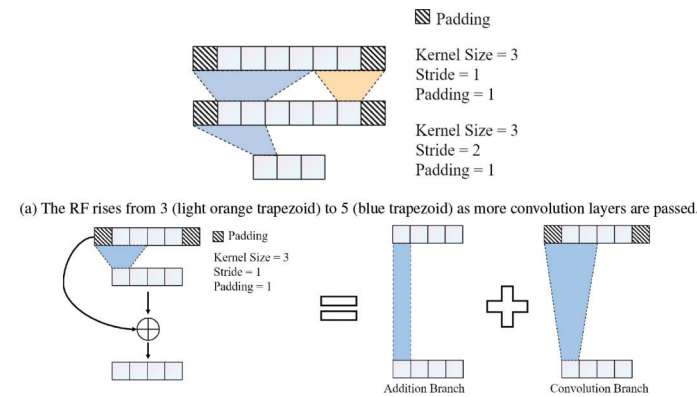


Figure 5: The receptive field (RF) of the model with two layers of convolution and the model with residual blocks.

RF is 3 after one layer of 3×3 convolution. The figure shows that the size of the RF increases as the number of layers of the CNN model network increases.

Figure 5b illustrates the case where the RF has different values on two different branches after using a residual block. A residual block can be seen as a module combining a branch with a 3×3 convolution and a branch with an addition operation. The value of the RF after 3×3 convolution is 3, while for the addition operation the RF is only 1.

In models with different branches, the maximum of the branches is used as the RF value. For example, in Figure 5b, we use the receptive fields of the 3×3 convolution instead of the residual block. The widely used CNN models usually consist of many connected convolutional layers, and the RF size calculated in the last layer of the model is usually larger than the resolution of the input image.

2.3 The Proposed Method

The method proposed in this paper is shown in Figure 4b. We first determine the number of inputs after tiling the input image (which is 16 in Figure 4b), and then decide a position for the activations to be aggregated, so that the CNN model is divided into two parts: the head and the tail.

The input image is tiled according to the receptive field, and the black grids on the input image in Figure 4b show the size of each tile obtained after tiling. RF-based tiling induces some overlap between adjacent tiles, as illustrated in Figure 4b. These overlaps are designed so as to ensure that the output of the network remains exactly the same.

These tiles are fed one after another into the head of the CNN model, and the activations obtained from each tile are aggregated and fed into the tail of the model to complete the remaining inference.

We feed the model smaller-sized inputs each time, reducing memory consumption peak. As shown in Figure 3, the peak memory consumption of the CNN model occurs at the Conv2 layer. The aggregated activations are guaranteed to be the same as the original un-tiled model, and, thus, RF-based tiling does not affect the model accuracy.

After the max-pooling operation with stride 2, the memory occupied by the activations is significantly reduced as the length and width of the activations are downsampled to one half of the original size. Max-pooling, however, results in an exponential increase in RF, which results in excess computation, and we will discuss the trade-off between memory and computation in the next section on the position of activation aggregation.

3 Computation vs. Memory Trade-off

Our proposed method consists of feeding to the network a sequence of overlapping adjacent tiles. Hence, the overlap between these tiles is fed into the model multiple times for computation, which introduces a computation redundancy overhead. For a particular CNN model, the computation/memory trade-off that is achieved depends on the specific layer at which the activations tiles are aggregated.

In this section, we explore the memory and computational overhead trade-offs for sequential convolutional neural networks and convolutional neural networks with residual blocks, presenting formulas for theoretically calculating the computation and memory trade-offs of CNN models.

Furthermore, we use a parameterizable CNN toy model to explore the influence of parameters on memory consumption and computation trade-offs, which provides a framework for designing more memory-friendly CNN models. We conclude this section with experiments on several successful CNN architectures to show the memory consumption and computation trade-offs they can achieve.

3.1 Sequential Convolutional Neural Networks

3.1.1 Peak Memory Consumption

First, we theoretically explore the peak memory consumption achieved by a sequential CNN model. Figure 6 illustrates the architecture of this sequential CNN model. We only study the optimization of two-dimensional convolutional neural networks because two-dimensional convolution is the basis of

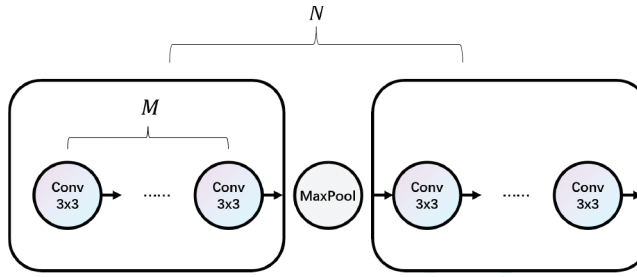


Figure 6: A sequential CNN model consisting of N blocks and M modules.

computer vision research. Successful optimization on two-dimensional convolutional neural networks can provide optimization theory for higher-dimensional convolutional neural networks.

We consider a network made of N blocks. In each block, there are M convolution layers. H_n^m, W_n^m, and C_n^m represent the height, width, and channel, respectively, of the input activations of the m-th convolution module in the n-th block. We use $M+1$ to denote the output of the M-th convolution module. Every two blocks are connected to each other using max-pooling.

The peak memory consumption O can then be calculated according to Equation (1). R is the parameters of model.

$$\begin{aligned} O &= \max\left(H_n^m W_n^m C_n^m + H_n^{m+1} W_n^{m+1} C_n^{m+1}\right) + R, \\ n &= 1, 2, \ldots, N, \quad m = 1, 2, \ldots, M \end{aligned} \tag{1}$$

Equation (1) can be applied to PyTorch and our proposed method. Equation (1) shows that the peak memory consumption for model inference occurs at a certain layer.

The maximum memory O_f required by the Tensorflow Lite naive method can be calculated using Equation (2). Tensorflow Lite native method allocates the memory required by the activations of all layers on the initialization. Therefore, the maximum memory consumption during inference is the sum of the memory required by all activations and includes the parameters R of the model.

$$O_f = \left(\sum_{n=1}^{N}\sum_{m=1}^{M} H_n^m W_n^m C_n^m\right) + R \tag{2}$$

3.1.2 Multiply–Accumulate Operation

Moreover, we discuss the MAC (the number of multiply-accumulate operations) required for the inference after using our proposed method. We use Q to denote the MAC required to use our method, and Q_f to denote the MAC for using the

methods (Tensorflow Lite Naive, Tensorflow Lite Optimized, PyTorch Naive, PyTorch BN Fused) in Figure 2 other than our proposed method.

The parameter k_n^m indicates the kernel size of the m-th convolutional layer in the n-th block. H_n^m, W_n^m, and C_n^m represent the height, width, and channel, respectively, of the input activations of the m-th convolution module in the n-th block. t denotes the number of tiled input images, and activations are aggregated after l block.

We can tile only the height axis or the width axis of the image, or both. Tiling both height and width make the tiles smaller and requires less memory, so we will only discuss the case where both height and width are tiled. The minimum value of t is 4, which means that we tile both height and width once. The size of the activations will be gradually reduced by the max-pooling layers. The maximum value of t is $H_{l+1}^1 W_{l+1}^1$, i.e., the activations before aggregation are tiled into H_{l+1}^1 by W_{l+1}^1 activations.

Q can be calculated using Equation (3). The first term expresses the amount of MAC required after tiles are fed into the head of the model, whereas the second term describes the MAC required by the execution of the tail of the model. Q_f can be calculated using Equation (4). Equation (4) has the same form as the second term of Equation (3), feeding the activations that are not tiled into the model.

$$\begin{aligned} Q = t \sum_{n=1}^{l} \sum_{m=1}^{M} H_n^m W_n^m \left(C_n^m\right)^2 \left(k_n^m\right)^2 + \sum_{n=l+1}^{N} \sum_{m=1}^{M} H_n^m W_n^m \left(C_n^m\right)^2 \left(k_n^m\right)^2, \\ t = 4, 9, 16, \ldots, (i+1)^2, \ldots, H_{l+1}^1 W_{l+1}^1, \quad i \in \mathbb{N} \\ l = 1, 2, \ldots, N-1 \end{aligned} \tag{3}$$

$$Q_f = \sum_{n=1}^{N} \sum_{m=1}^{M} H_n^m W_n^m \left(C_n^m\right)^2 \left(k_n^m\right)^2 \tag{4}$$

3.2 Residual Convolutional Blocks

3.2.1 Peak memory consumption

Similar to 3.1, we assume that there are N residual blocks, and each block has M convolution modules, as shown in Figure 7. The peak memory consumption O_r of the residual block after using our proposed method can be calculated using Equation (5).

Note that because the residual structure needs to add the input activations and output activations, the input activations $H_n^1 W_n^1 C_n^1$ will always stay in the device memory before completing the last step.

In this paper, we only consider the aggregation of activations outside the residual block, although we can also do it inside the residual block since the residual block requires the addition operation. It is necessary to keep the

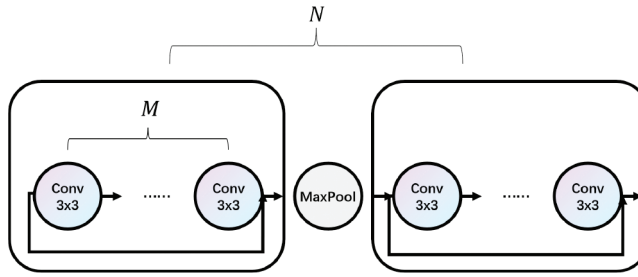


Figure 7: A Residual CNN model consisting of N blocks and M modules.

size of the added activations the same, which will increase extra memory consumption.

The memory consumed by the Tensorflow Lite naive method can still be calculated using Equation (2) and PyTorch memory consumption can be calculated using Equation (5).

$$O_r = H_n^1 W_n^1 C_n^1 + \max\left(H_n^m W_n^m C_n^m + H_n^{m+1} W_n^{m+1} C_n^{m+1}\right) + R \\ n = 1, 2, \ldots, N, \quad m = 1, 2, \ldots, M \tag{5}$$

3.2.2 Multiply–Accumulate Operation

In fact, the residual structure does not change the way MAC is calculated after using this method. We can use Equation (3) to calculate the MAC of the residual model. In addition to our method in Figure 2, MAC can be calculated using Equation (4).

3.3 Guidelines for Designing a More Memory-Friendly Model

In this section, we use a parametrizable toy model to analyze the trade-off between memory and computational overhead. This analysis can provide us with tools to design memory-friendly CNN models.

The parameterizable toy model architecture is shown in Figure 8. The toy model is mainly composed of residual blocks, and it consists of N blocks, which are connected by max-pooling layers. Max-pooling layers downsample both the height and width of the activations by half. The output channel C_o of each module is double that of input channel C_i of the module. C_{st} is denoted as the number of channels of the first convolution. In each block, B residual blocks are connected, and each residual block is composed of M 3×3 convolutional layers. In the last residual block of the module, a 1×1 convolutional layer is used to increase the number of output channels.

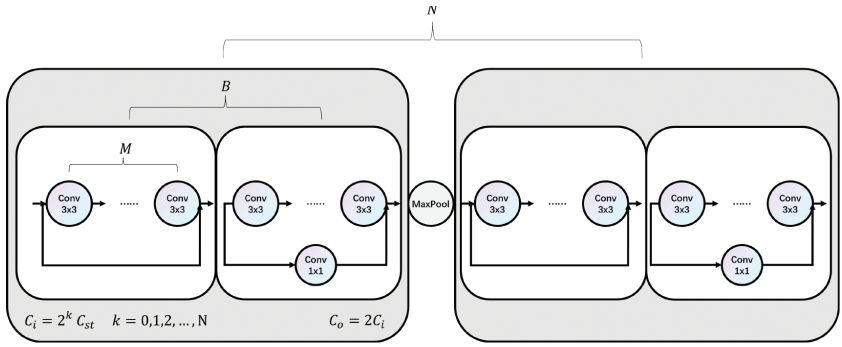


Figure 8: A parameterizable toy CNN model.

3.3.1 Memory Footprint

Figure 9a to 9d show the memory footprints of the toy model with $N = 3$, $B = 2$, $M = 3$, and $C_{st} = 32$ using the PyTorch naive implementation and our proposed method with 16 tilings. We represent the toy model with $N = 3$, $B = 2$, $M = 3$, and $C_{st} = 32$ as toy model 1.

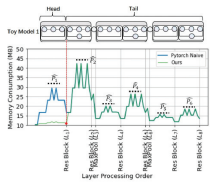
The input size of the model is 224×224. The x-axis represents the order of model execution. L_1 to L_8 represent the positions where the activations can be aggregated using our proposed method. The positions where the activations are aggregated are marked by red dots. The y-axis shows the memory consumption. The maximum memory consumption for each residual block is denoted by $\hat{P}_1$ to $\hat{P}_6$.

Similarly, Figure 9e to 9h show the memory footprint of the toy model with $N = 3$, $B = 2$, $M = 4$, and $C_{st} = 32$, and the aggregated positions are also denoted by L_1 to L_8. We represent the toy model with $N = 3$, $B = 2$, $M = 4$, and $C_{st} = 32$ as toy model 2. The maximum memory consumption for each residual block is denoted by P_1 to P_6. Figure 9e to 9h using the PyTorch naive method consumes more memory than Figure 9a to 9d and the y-axis rises from 15 MB. P is higher than $\hat{P}$ for the same subscript. This is because the increased convolution parameters take up some memory, but the memory required for activations does not increase with M (the highest point on the y-axis minus the lowest point).

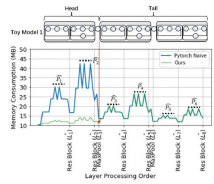
3.3.2 Memory-Computation Trade-off

We will now describe the memory and computation trade-off using our method and the PyTorch naive method in this part.

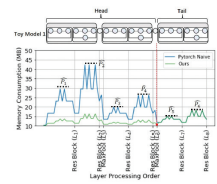
Figure 9i shows the change in the MAC relative overhead ratio for increasing tiling size. The y-axis indicates the relative MAC improvement at different aggregation positions.



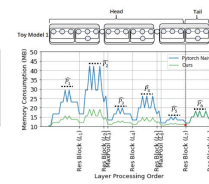
(a) Comparison of memory consumption in the toy model 1 using our method (aggregated at L_1) and the PyTorch naive method.



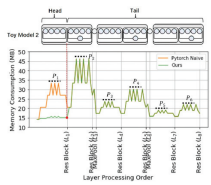
(b) Comparison of memory consumption in the toy model 1 using our method (aggregated at L_3) and the PyTorch naive method.



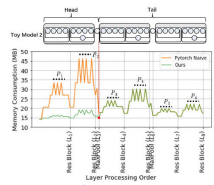
(c) Comparison of memory consumption in the toy model 1 using our method (aggregated at L_6) and the PyTorch naive method.



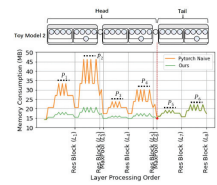
(d) Comparison of memory consumption in the toy model 1 using our method (aggregated at L_7) and the PyTorch naive method.



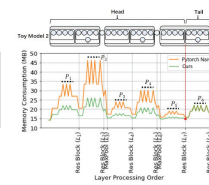
(e) Comparison of memory consumption in the toy model 2 using our method (aggregated at L_1) and the PyTorch naive method.



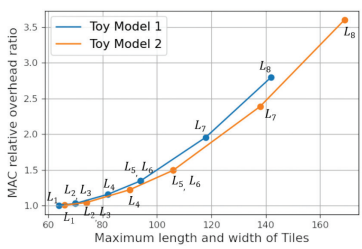
(f) Comparison of memory consumption in the toy model 2 using our method (aggregated at L_3) and the PyTorch naive method.



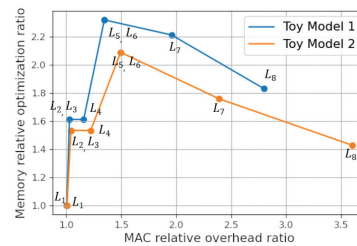
(g) Comparison of memory consumption in the toy model 2 using our method (aggregated at L_6) and the PyTorch naive method.



(h) Comparison of memory consumption in the toy model 2 using our method (aggregated at L_7) and the PyTorch naive method.



(i) The relationship between tiling size and computational redundancy overhead.



(j) The relationship between computational redundancy overhead and memory optimization.

Figure 9: Analysis of parametric toy models in terms of computational redundancy overhead and memory trade-off.

The MAC of the model using our proposed method can be calculated using Equation (3), where $t = 16$ and l depends on the aggregation position (from 1 to 8). The MAC of the PyTorch method can be calculated using Equation (4). $y = 1$ indicates that the same MAC is needed to use our proposed method and the PyTorch naive method. The blue curve is for the toy model 1. and the orange curve is for the model B.

We tiled the 224×224 input into 16 parts based on the receptive field. Although, the size of each activation before aggregation inside the model is the same, the size of the tiles mapped to the input image is different for different positions of activation. For the 16 input tiles, the tile size in the middle part is

the largest, and the smallest size tile is at the edge of the image. The largest tile is the rectangle with equal length and width. The largest tile causes the largest memory bottleneck. For the sake of calculation, we assume that all 16 tiles are of the same size as the largest tile. Therefore, we choose the largest size tile as the x-axis of Figure 9i.

Figure 9j shows the memory relative optimization ratio as the MAC overhead ratio increases. The y-axis of Figure 9i shares the x-axis of Figure 9j. The y-axis of Figure 9j represents the peak memory consumption using the PyTorch naive method divided by the peak memory consumption using our method. The $y = 1$ indicates that the peak memory consumption using our proposed method is as high as the PyTorch naive method. The memory consumed using our method and the memory consumed by PyTorch can be calculated using Equation (5).

As the aggregation position deepens, more MAC is required using our proposed method. The relative memory optimization rate does not increase with deeper aggregation positions.

In the following, we summarize four cases where the memory optimization changes as the computational redundancy (aggregation position) varies. We use Figure 9a to 9d as examples.

(1) The memory consumption using our proposed method is equivalent to the memory consumption of PyTorch (aggregation position: L_1):

This happens when the aggregation position is before the peak memory consumption in the PyTorch naive method, as shown in Figure 9a. $\hat{P}_2$ is the peak memory consumption in the PyTorch naive method, and when the aggregation position is in L_1, the activations after aggregation will still increase to $\hat{P}_2$ without causing memory optimization.

(2) The memory relative optimization ratio does not increase as the aggregation position deepens (aggregation position: from L_2 or L_3 to L_4):

This case is similar to the first case, and occurs when the memory consumption at the head of the model is smaller than the memory consumption at the tail of the model. When the activations are aggregated at L_2, L_3, and L_4, although $\hat{P}_2$ is crossed, $\hat{P}_4$ becomes the new memory consumption peak, as shown in Figure 9b. This change in position does not make the memory consumption more optimal.

(3) The memory relative optimization ratio goes up (aggregation position: from L_1 to L_6):

This is what we expect to happen. This situation occurs across the current memory consumption peak. For example, when our aggregation positions go from L_1 to L_2 or L_3, we cross $\hat{P}_2$, and optimization of

memory consumption increases. When we cross $\hat{P}_4$ from L_4 to L_5 or L_6, the optimization of memory consumption will also increase.

(4) The memory relative optimization ratio decreases (aggregation position: from L_5 or L_6 to L_7):

This happens when the memory consumed at the head of the model is larger than at the tail of the model. As the aggregation position deepens, the tiling increases, and the memory consumption of the input tiling in the head of the model becomes larger and larger in this case. For example, when going from L_5 or L_6 to L_7, we may naturally think of $\hat{P}_6$ as the memory bottleneck. But at this time, the memory consumption in the model head is already higher than $\hat{P}_6$ as the tiling increases, as shown in Figure 9c and 9d.

3.3.3 Memory-friendly model guideline

We give three guidelines for designing memory-friendly models that are well suited for RF-based tiling memory optimization:

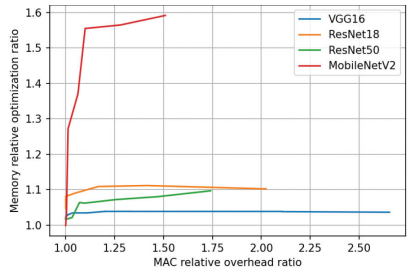
(1) The maximum memory consumption of the model tends to occur at the relatively low layers. This way there is only a small amount of computational redundancy when aggregating the activations after crossing the memory consumption peak.

(2) The memory used by the activations should be higher than the memory used by the parameters, so that the bottleneck will be better reflected in the memory occupied by the activations. This makes better memory optimization possible with our method.

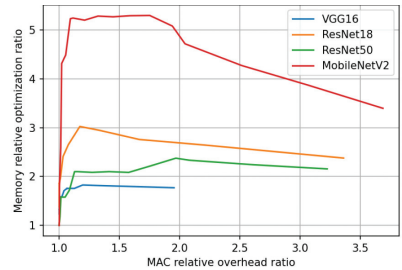
(3) The number of convolutions in a residual block should be reduced appropriately because the method cannot aggregate activations in the residual block. The increase in the number of convolutions in the residual block will lead to more computational redundancy when aggregating activations behind the residual block and will result in more parameters.

4 Results and Discussion

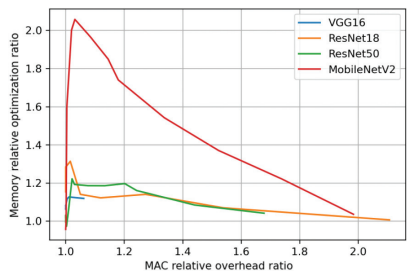
We conducted experiments with our proposed method using some successful CNN models in an experimental environment with the PyTorch [13] deep-learning framework and Nvidia GPUs. Note that the optimization of device memory in the CNN inference phase depends to some extent on the optimization capabilities of the framework.



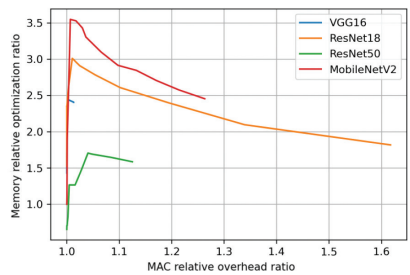
(a) Input: 224×224, Tiling: 16



(b) Input: 1024×1024, Tiling: 16



(c) Input: 224×224, Tiling: 4



(d) Input: 1024×1024, Tiling: 4

Figure 10: Computation and memory trade-offs at different resolutions input after applying our proposed method to some widely used CNN models.

Figure 10 shows the effect of different input resolutions and the different number of tiles on the memory-computation trade-offs after using our method. We can see from Figure 10 that MobileNetV2 achieves a higher memory optimization rate than other models. Because MobileNetV2 is a lightweight network, the memory occupied by the parameters is much smaller than the memory occupied by the activation. In Figure 10b, we see that MobileNetv2 can achieve a memory optimization ratio of 5.3, which means that the memory burden is only 19% that of the original one after using our method.

A larger input resolution results in a higher memory optimization ratio. For example, MobileNetV2 achieves at most a 1.6× memory optimization ratio in Figure 10a, but 5.3× memory optimization ratio in Figure 10b. Our proposed method is more suitable for optimizing memory bottlenecks resulting from high-resolution inputs.

Unfortunately, a larger amount of tiling may require more computational overhead to achieve the same memory optimization ratio using smaller tiling sizes. In Figure 10a MobileNetV2 requires 1.5× the computational overhead to achieve a memory optimization ratio of 1.6×. However, in Figure 10c, only less than 1.1× computational overhead is needed to achieve a 2.0× memory

optimization ratio. We recommend selecting the input size, the number of tiles, and the aggregation position of the activations according to the usage requirements.

In both figures, we can find that the optimization rate of memory rises quickly with little computational overhead, which indicates that the memory bottleneck occurs at the lower layers when these models are performing default inference. The curves of both MobileNetV2 and ResNet50 in Figure 10a do not decrease, which indicates that the memory bottleneck occurs at the tail of the model.

5 Conclusion

In this paper, we propose a method to reduce the memory consumption of convolutional neural networks in the inference phase that is orthogonal to other model compression methods. Using our method and the analytical results of this formulation, we can design a more memory-friendly model. We have used our proposed method in some successful CNN models, resulting in a reduction in the memory consumption of the model in the device. However, our proposed method also imposes a certain computational overhead, showing that it is more effective for high-resolution input images.

References

[1] A. Araujo, W. Norris, and J. Sim, "Computing Receptive Fields of Convolutional Neural Networks," *Distill*, 4(11), 2019, e21.

[2] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, 1251–8.

[3] A. Gural and B. Murmann, "Memory-Optimal Direct Convolutions for Maximizing Classification Accuracy in Embedded Applications," in *International Conference on Machine Learning*, 2019, 2515–24.

[4] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, 2015.

[5] S. Han, J. Pool, J. Tran, and W. Dally, "Learning Both Weights and Connections for Efficient Neural Network," *Advances in Neural Information Processing Systems*, 28, 2015, 1135–43.

[6] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv preprint arXiv:1503.02531*, 2015.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-Level Accuracy with 50× Fewer Parameters and <0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[9] J. Jin, A. Dundar, and E. Culurciello, "Flattened Convolutional Neural Networks for Feedforward Acceleration," *arXiv preprint arXiv:1412.5474*, 2014.

[10] R. Krishnamoorthi, "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[11] J. Lee and Y. Pisarchyk, "Optimizing TensorFlow Lite Runtime Memory," 2020, URL: https://blog.tensorflow.org/2020/10/optimizing-tensorflow-lite-runtime.html.

[12] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet v2: Practical Guidelines for Efficient CNN Architecture Design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, 116–31.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An Imperative Style, High-Performance Deep Learning Library," *arXiv preprint arXiv:1912.01703*, 2019.

[14] Y. Pisarchyk and J. Lee, "Efficient Memory Management for Deep Neural Net Inference," *arXiv preprint arXiv:2001.03288*, 2020.

[15] C. F. Sabottke and B. M. Spieler, "The Effect of Image Resolution on Deep Learning in Radiography," *Radiology: Artificial Intelligence*, 2(1), 2020, e190015.

[16] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetv2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, 4510–20.

[17] X. Tao, D. Zhang, W. Ma, X. Liu, and D. Xu, "Automatic Metallic Surface Defect Detection and Recognition with Convolutional Neural Networks," *Applied Sciences*, 8(9), 2018, 1575.

[18] T. Wang, Y. Chen, M. Qiao, and H. Snoussi, "A Fast and Robust Convolutional Neural Network-Based Defect Detection Model in Product Quality Control," *The International Journal of Advanced Manufacturing Technology*, 94(9), 2018, 3465–71.

[19] Y. Wen, A. Anderson, V. Radu, M. F. O'Boyle, and D. Gregg, "TASO: Time and Space Optimization for Memory-Constrained DNN Inference," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, 199–208.

[20] S. Wu, M. Zhang, G. Chen, and K. Chen, "A New Approach to Compute CNNs for Extremely Large Images," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, 39–48.

[21] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, 6848–56.