Variable batch size across layers for efficient prediction on CNNs

Anamitra Roy Choudhury*, Saurabh Goyal*, Yogish Sabharwal *, Ashish Verma†
*IBM Research - India, New Delhi
{anamchou, saurago1, ysabharwal}@in.ibm.com
† IBM T. J. Watson Research Center, Yorktown Heights, New York
ashish.verma1@ibm.com

***Abstract*—CNNs are used extensively for computer vision tasks like activity recognition, image classification, segmentation etc. The large compute memory required in these applications restricts the use of high batch size during inference, thereby increasing the overall prediction time. Prior work addresses this issue through various model compression mechanisms like weight/filter pruning, quantizing the parameters/intermediate outputs, etc. We propose a complementary technique where we improve inference time by using variable batch sizes (VBS) across the layers of a CNN. This optimises the memory-time trade-off for each layer and leads to better network throughput. Our approach does not make any modifications to the existing network (unlike pruning or quantization techniques) and thus there is no impact on the model accuracy. We develop a dynamic program (DP) based algorithm that takes inference time and memory required by different layers of the network as input, and computes the optimal batch sizes for each layer depending on the available resources (RAM, storage space etc.). We demonstrate our findings in two different settings: video inference on K80 GPUs and image inference on Edge devices. On video networks like C3D, our VBS algorithm gives up to 61% higher throughput compared to a fixed batch size baseline. On image networks like GoogleNet, ResNet50 etc., we achieve up to 60% higher throughput compared to a fixed batch size baseline.**

***Keywords*-convolutional neural network; inference; batch;**

I. Introduction

Convolution Neural Networks (CNNs) have become the most sought after machine learning models for various computer vision tasks like image classification, semantic segmentation, activity recognition etc., due to their unparalleled performance. However, often these models tend to be highly demanding in terms of their computation and storage memory due to the large number of parameters. For example, image classification networks like Inception-v4 [1] consist of 43M, object localization models like Yolo [2] consist of 270M parameters, image segmentation models like FCN16 [3] employ 135M parameters, while video activity recognition models like C3D have 79M parameters [4].

Prior work has considered techniques like weight/filter pruning [5, 6, 7], parameter quantization [8, 9, 10, 11], low-bit precision arithmetic [12, 13], low-rank decomposition [14, 15, 16, 17, 18] etc., to improve the efficiency of these network architectures, especially for resource constraint devices like mobile phones, edge devices and wearable devices. Another way to improve the efficiency of CNNs is to design networks by optimizing the inference time, compute memory, energy utilization etc. while maintaining the accuracy. Networks like SqueezeNet [19] and MobileNet [20] restrict their kernel sizes to 1x1 and 3x3 to reduce the compute and memory requirements and inference faster.

In this paper, we focus on a complementary aspect of the efficiency, viz, batch inference. Batch inference is more efficient than single image inference as the cost of fetching the model gets amortized over the input batch, leading to better utilization of resources and an increase in the throughput. Batch inference is particularly useful when the vision tasks are carried out on resource constrained mobile phones or wearable devices e.g. Google glasses [21]. Also, for edge devices used in Internet of Things (IoT) applications (like Arduino Boards fitted with low powered camera sensor ESP8266 that are used for analysis of soil quality and texture on agricultural fields), getting faster inference becomes critical as it reduces the power consumption and increases battery life.

Another application for batched inference is in a cloud environment where inference requests for different clients/models are served. In such a setting, the cost is determined by the utilization of various resources such as memory and compute nodes. Two common techniques to reduce the costs are to (i) improve the inference throughput and (ii) load multiple models into the memory of every node. While the former approach is beneficial for models that are very frequently used (thus multiple requests can be combined), the latter is favourable for less frequently used models. In practice, the frequency of model usage lies between the two extremes and hence both the techniques prove valuable. Improving the inference throughput under conditions of constrained memory utilization is ideal for such scenarios.

Currently, the deep learning frameworks like Pytorch, Caffe2, TensorFlow etc. work with a predefined batch size that remains same across layers. The maximum batch size possible is restricted by the computational memory required by the model on a given device. In this paper, we propose an alternative paradigm where instead of processing every layer with the same batch size we use different batch sizes for different layers depending on their computational requirement. Two key observations that motivate our idea of variable

2159-6190/20/$31.00 ©2020 IEEE
DOI 10.1109/CLOUD49709.2020.00065

batch size are (i) the computation memory requirement differs significantly across layers of a network due to varying input/output sizes at different layers; (ii) the computational efficiency of processing a layer improves with increasing batch size (c.f. Section II). Thus, different layers can be computed with variable batch sizes (VBS) resulting in an overall faster inference and increased throughput. There has been some work to use different batch sizes for making the training more efficient [22, 23]. In [22], the authors experimentally determine smaller batch sizes to use for a group of consecutive layers that lead to better on-chip memory utilization. In [23], the authors consider breaking down the batch at a given layer into smaller batches in order to use more memory efficient cuDNN algorithms. However, both these methods require all the output activations to be stored at every layer; they do not use variable batch sizes to reduce the overall memory utilization due to storage of output activations and are thus orthogonal to the techniques discussed in this paper.

We propose a novel dynamic program based algorithm that takes inference time and memory required by different layers of the network as input, and determines the optimal batch sizes for each convolution layer of a CNN. This optimises the memory-time trade-off for each layer and hence leads to better inference time and higher network throughput. We evaluate our model on state-of-the-art video and image classification networks such as C3D, GoogleNet [24], ResNet50 [25], SqueezeNet-v0 and MobileNet-v1 and show that our approach achieves up to 60% performance improvement in inference throughput.

Our Contribution: : The focus of this paper is to study optimization techniques for CNNs complementary to the existing ones. Our main contributions are as follows:

- We improve inference time using different batch sizes for different layers of a CNN. We propose a novel dynamic program based algorithm to determine the optimal batch size under given memory constraints that maximizes the inference throughput (or minimizes inference time).
- We evaluate our model on state-of-the-art models such as C3D, GoogleNet, ResNet50, SqueezeNet-v0 and MobileNet-v1 and show that our approach achieves up to 60% performance improvement in inference throughput.

Some additional advantages (other than faster inference) of our proposed scheme over other optimization techniques are: i) we do not make any modifications to the existing network unlike pruning or quantization techniques and therefore there is no impact on model accuracy. ii) our proposed optimization can be implemented along with other model compression techniques mentioned earlier. iii) our optimization technique is dynamic in nature and doesn't require prior knowledge about system specification and load to reduce the inference time. Thus, it is easier than designing a new model every time to fit the memory constraints.

The rest of the paper is organized as follows. In Section II, we describe our main idea of variable batch size. Section III presents our dynamic program and algorithm to compute the optimal batch sizes for efficient inference. Section IV presents our experimental results with different models. Finally, we present our conclusions in Section V.

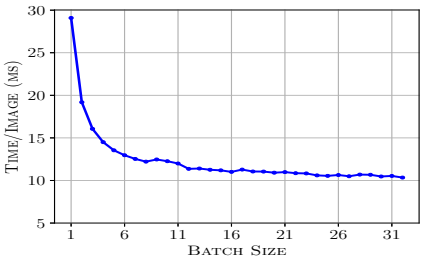


Figure 1. Inference time per image for different batch sizes for ResNet-50 (values decrease with increasing batch size)

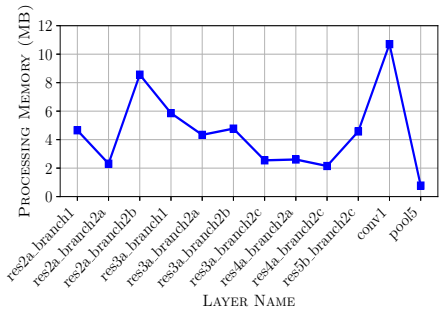


Figure 2. Memory requirement to process different layers of ResNet50.

II. Variable Batch Sizing

We now describe the main intuition for variable batch sizing. The first key observation is that larger batch sizes improve the inference throughput. This is primarily due to the fact that the underlying kernels for convolution layers and fully connected layers comprise of vector and matrix operations that are more efficient for larger sizes. Figure 1 shows the variation of inference time (per image) with different batch size inputs for ResNet50 model. Inferencing with larger batch sizes however results in larger runtime memory which may not be always available. Thus, it is important to maximize the throughput while keeping the memory footprint within available memory limits.

The second key observation is that the memory requirement during inference varies with the layers even for a fixed batch size. Figure 2 shows total memory requirement for batch size=1 (input activation + output activation + working space) for the different layers of a ResNet50 model. Clearly, we see a lot of variation in the memory requirement across layers (from 1.07MB to 10.7MB). For example, an inferencing code for ResNet-50 network requires 10.7MB of memory (sum of sizes of input activations, output activations and working space) for conv1 layer, whereas the memory requirement

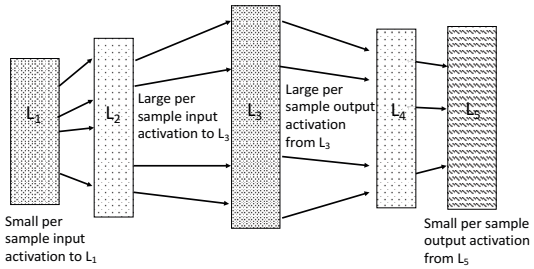


Figure 3. Variation in input and output sizes across different layers of a network motivates the scheme of variable batch sizes.

L_i	ith layer of the network
$\text{time}(i,b)$	per-sample time to process L_i with batch size b
$\text{in}(i,b)$	activation memory for b input for L_i
$\text{out}(i,b)$	activation memory for b output for L_i
$\text{ws}(i,b)$	temporary memory to process b input samples at L_i
$\langle i,b,mem\rangle$	configuration made by L_i, batch size b and memory mem
$\texttt{OPT}[i,j,b,mem]$	optimal per-sample time to process b batch size from L_i to L_j each $L_i,\ldots,L_j$ computed with batch size at most b
$\texttt{OPTExact}[i,j,b,mem]$	optimal per-sample time to process b batch size from L_i to L_j each $L_i,\ldots,L_j$ computed with batch size exactly b

Table I
NOTATIONS USED IN THE PAPER.

is much less (around 2.3MB) for res2a_branch2a layers. Hence in a resource constrained system, one layer may allow computations using a larger batch size, while another layer may not. We observed similar kind of variations in other models as well, where typically the first few layers (towards input) dominate the memory requirement. We leverage this characteristics of CNNs and propose a novel technique to make the batch inference more efficient.

Let us consider an arbitrary network (see Figure 3) consisting of a sequence of 5 layers, L_1, $L_2 \cdots L_5$, such that output of L_i feeds to its successor layer L_{i+1}. Thus, forward pass during inference involves computation at each of the 5 layers in sequence starting with input to layer 1 and determining output activations at layer 5. Now suppose that the input and output sample activation sizes for layers L_1 and L_5 are small and that of layer L_3 are large. Thus, with uniform batch size, the memory required by layer L_3 restricts the batch size that can be processed by the network. However, by using a larger batch size for layers L_1 and L_5, we can benefit from the efficiency of performing larger matrix/vector operations. For e.g., suppose that L_1 and L_5 can process a batch size of b whereas L_3 can only process a batch size of $b' < b$. For simplicity, assume b' divides b. Then we can process L_1 with a batch size of b producing output activations of b samples at layer L_1. This is followed by b/b' phases wherein in each phase, layers L_2-L_4 are processed with a batch size of b' at a time ; at the end of these phases, activations of b samples are available as an input for layer L_5. Finally, we process L_5 with a batch size of b. Note that at any point of time input and output activations for only b' samples are required to be stored for layer L_3.

Thus, using different batch sizes for the layers, one is expected to achieve a better throughput than that obtained by working with uniform batch size. However, figuring out the optimal batch size for different layers to maximize throughput is non-trivial. This is discussed in the next section.

III. DYNAMIC PROGRAMMING (DP) ALGORITHM

In this section, we present our dynamic programming algorithm to determine the optimal batch size to be used at each layer that minimizes the inference time under the specified memory constraint. We first consider the simpler case of linear structured CNNs comprising of n layers $L_1, \ldots L_n$, wherein the output of any layer, L_i directly feeds into the next layer L_{i+1}. DNNs like AlexNet, MobileNet-v1 and VGGNet-16 belong to this class of networks. We next consider the scenario where the layers of the model form branches within a path. DNNs like GoogleNet, SqueezeNet and ResNet are encompassed in this group.

Linear structured models: We first consider the simpler case of linear structured CNNs. Let $\text{time}(i,b)$ denote the per-sample time for processing layer L_i with batch size b (this can be obtained by processing layer L_i with a batch size of b and dividing the time taken by b). The notations $\text{in}(i,b)$ and $\text{out}(i,b)$ denote the memory required to store activations for b input and output samples respectively for layer L_i. Further, let $\text{ws}(i,b)$ denote the temporary (extra) memory required by layer L_i to complete its processing for b input samples. A *configuration* is a tuple $\langle i,b,mem\rangle$, where i denotes the layer L_i, b denotes a batch size and mem denotes the amount of memory. We say that a configuration $\langle i,b,mem\rangle$ is *feasible* if the total memory required for performing computations at layer L_i with a batch size of b is within mem units of available memory, i.e., $\text{in}(i,b)+\text{ws}(i,b)+\text{out}(i,b) \leq mem$.

We maintain two DP tables, $\texttt{OPT}[\cdot,\cdot,\cdot,\cdot]$ and $\texttt{OPTExact}[\cdot,\ \cdot,\cdot,\cdot]$. An entry $\texttt{OPT}[i,j,b,mem]$ ($\texttt{OPTExact}[i,j,b,mem]$ resp.) stores the optimal per-sample time to do inference of a batch of size b from layer L_i to layer L_j of the CNN, wherein each of the layers $L_i,\ldots,L_j$ is computed using batch size *at most* (*exactly*) b. Note that it is possible for $\texttt{OPT}[i,j,b,mem]$, that all the layers L_i, ..., L_j are computed with batch sizes strictly < b. In both the above definitions, the layers $L_i,\ldots,L_j$ use a total of at most mem units of memory including that required for storing the activations for input samples of layer L_i and the output samples of layer L_j. Table I lists all the relevant notations used.

Let us first consider the computation of entry $\texttt{OPTExact}[i,j,b,mem]$. Suppose in the optimal solution, layer L_k, $i \leq k \leq j$ is computed with batch size b. Then the total time per sample to compute layers L_i to L_j in this scenario can be expressed as the sum of three quantities:

(i) time per sample to compute layers L_i to L_{k-1} using batch size $\leq b$ and memory $= mem$.
(ii) time per sample to compute layer L_k with batch size b

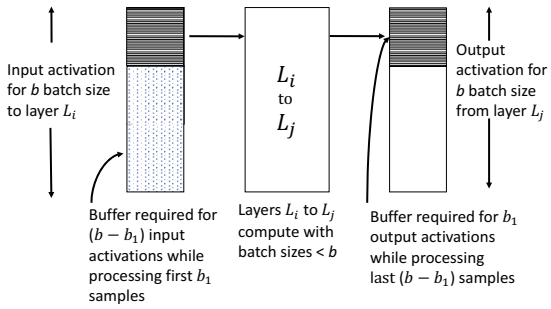


Figure 4. Input and output activations to be buffered when a layer processes b samples in using small batches.

and memory $= mem$. (finite if $\langle k, B, mem\rangle$ is feasible)

(iii) time per sample to compute layers L_{k+1} to L_j using batch size $\leq b$ and memory $= mem$.

As we don't know the layer L_k, we consider every layer between L_i and L_j and choose the layer L_k that gives us the best solution. This yields the following recurrence:

$$\texttt{OPTExact}[i, j, b, mem] = \min_{i \leq k \leq j} \left\{ \begin{array}{c} \texttt{OPT}[i, k-1, b, mem] \\ + \texttt{OPTExact}[k, k, b, mem] \\ + \texttt{OPT}[k+1, j, b, mem] \end{array} \right\}. \quad (1)$$

$$\texttt{OPT}[i, j, b, mem] = (1/b) \cdot \min_{1 \leq b_1 \leq b} \left\{ \begin{array}{l} b_1 \cdot \texttt{OPTExact}[i, j, b_1, mem - \text{in}(i, b - b_1)] + \\ (b - b_1) \cdot \texttt{OPT}[i, j, b - b_1, mem - \text{out}(j, b_1)] \end{array} \right\} \quad (2)$$

Note that the use of OPT permits layers L_i to L_{k-1} and layers L_{k+1} to L_j to be processed with a batch size smaller than b. To handle boundary cases, when $k = i$ or $k = j$, we take $\texttt{OPT}[i, j, b, mem] = 0$ whenever $i > j$.

Let us now consider the computation of entry $\texttt{OPT}[i, j, b, mem]$. Suppose that the optimal time to process a batch size b from layer L_i to L_j is obtained by breaking it down into smaller batches of size $b_1, b_2, \ldots, b_w$ where $\sum_{u=1}^{w} b_u = b$. Then the total time to compute layers L_i to L_j can be expressed as the sum of two quantities:

(i) time to compute layers L_i to L_j using batch size b_1.

(ii) time to compute layers L_i to L_j using one or more batch sizes summing up to $b - b_1$.

As we don't know the batch size b_1 used in processing layers L_i to L_j, we consider every non-zero value of b_1 less than or equal to b. The second component (time for batch sizes adding up to $b - b_1$) can be computed recursively. Thus,

Note the accounting of memory in the above recurrence. During the processing of the For the first batch of size b_1, the last $(b - b_1)$ samples are yet to be processed and hence consume a memory of $\text{in}(i, b - b_1)$ at layer L_i. Subsequently, for the processing of the remaining $b - b_1$ samples, the activations of the first b_1 processed samples occupy a memory of $\text{out}(j, b_1)$ at layer L_j (see Figure 4). The memory for the input and output activations for the samples processed in any phase are accounted in the call to OPTExact itself and hence do not require any adjustment in the available memory.

In order to compute the entries of OPT and OPTExact, we first compute all the values of $\text{in}(i, b)$, $\text{out}(i, b)$, and $\text{ws}(i, b)$. Note that all these can be statically computed. We next compute all the entries $\text{time}(i, b)$. These require a run through each layer with the corresponding batch size. All these entries can be computed once for a given model. For the base case,

$$\texttt{OPTExact}[i, i, b, mem] = \left\{ \begin{array}{ll} \text{time}(i, b), & \text{if } \langle i, b, mem\rangle \text{ is feasible} \\ \infty, & \text{otherwise} \end{array} \right. \quad (3)$$

$\texttt{OPT}[i, i, b, mem]$ is computed using recurrence (2). We also set $\texttt{OPT}[i, j, b, mem] = 0$ for all $i > j$. We then compute the entries of OPTExact and OPT using recurrence (2) where the starting and ending layers differ by d for $d = 1$ to $n - 1$.

The required optimal solution for inference on b samples with available memory mem is finally obtained from the entry $\texttt{OPT}[1, n, b, mem]$. It is straightforward to keep track of the optimal choice at each step in order to determine the optimal profile of batch sizes to be used at each layer corresponding to the optimal solution. Algorithm 1 presents the DP pseudocode.

Illustration with an example. Consider a three layer network L_1,L_2,L_3 such that $\text{in}(i, b) = \text{out}(i, b) = b$, for all layers; $\text{ws}(1, b) = \text{ws}(3, b) = b$, $\text{ws}(2, b) = 4b$; total memory available=7; $\text{time}(i, 1) = 4$ and $\text{time}(i, 2) = 3$ for all i. (i.e., the time to process 2 images at any layer is 6). Baseline can process only 1 image (as 2 images at L_2 need memory $2 \times (1 + 4 + 1) = 12 > 7$); thus, time to process 1 image is $\sum_i \text{time}(i, 1) = 12$. Our approach will process 2, 1 and 2 images at L_1, L_2 and L_3 respectively (with 2 rounds for 2 images at L_2). Thus, processing time per image is $(6+8+6)/2 = 10$. Let us see how the dynamic programming algorithm determines this.

Initialization (using Equation 3): For $i = 1$, 2, 3,
$\texttt{OPTExact}[i, i, 1, m] = 4$, for $3 \leq m \leq 7$, and ∞ else.
$\texttt{OPTExact}[i, i, 2, m] = 3$, for $6 \leq m \leq 7$, and ∞ else.
$\texttt{OPT}[2, 2, 1, m] = 4$, for $m = 6$, 7 and ∞ else.
$\texttt{OPT}[i, j, b, m] = 0$ for $i > j$.

Dynamic program computations for optimal solution are:

$$\begin{array}{rcl} \texttt{OPT}[1, 3, 2, 7] & = & \texttt{OPTExact}[1, 3, 2, 7] \\ & & \quad (b_1 = b = 2 \text{ in Equation } 2) \\ & = & \texttt{OPT}[1, 0, 2, 7] + \texttt{OPTExact}[1, 1, 2, 7] \\ & & \quad + \texttt{OPT}[2, 3, 2, 7] (k = 1 \text{ in Equation } 1) \\ & = & 0 + 3 + \texttt{OPT}[2, 3, 2, 7] \end{array}$$

Algorithm 1 Dynamic programming (DP) algorithm to compute individual layer batch size that maximizes throughput

Input: $\text{in}(i,b)$, $\text{out}(i,b)$, $\text{ws}(i,b)$ and $\text{time}(i,b)$, for each layer i and batch size b, TOT (total available memory).
Output: lookup$[i]$ which gives the list of batch sizes to be used for each layer i in the optimal solution.
Auxiliary data structures: OPT, OPTExact.
For each layer i, batch size b and available memory unit mem, compute $\texttt{OPTExact}[i,i,b,mem]$ using

$$\texttt{OPTExact}[i,i,b,mem] \quad = \quad \begin{cases} \text{time}(i,b), & \text{if } \langle i,b,mem\rangle \text{ is feasible} \\ \infty, & \text{otherwise} \end{cases}$$

Set $\texttt{OPT}[i,j,b,mem] \leftarrow 0$ for all $j < i$.
For each layer i, batch size b and available memory unit mem, compute $\texttt{OPT}[i,i,b,mem]$ using

$$\texttt{OPT}[i,j,b,mem] = (1/b)\cdot \quad \min_{1\le b_1\le b} \left\{ \begin{array}{l} b_1 \cdot \texttt{OPTExact}[i,j,b_1,mem - \text{in}(i,b-b_1)] + \\ (b-b_1)\cdot \texttt{OPT}[i,j,b-b_1,mem-\text{out}(j,b_1)] \end{array} \right\}$$

for $d = 1$ to $n-1$ **do**
  for $i = 1$ to $n-d$ **do**
    For each batch size b and available memory unit mem, compute $\texttt{OPTExact}[i,i+d,b,mem]$ using

$$\texttt{OPTExact}[i,j,b,mem] \quad = \quad \min_{i\le k\le j} \left\{ \begin{array}{l} \texttt{OPT}[i,k-1,b,mem] \\ +\ \texttt{OPTExact}[k,k,b,mem] \\ +\ \texttt{OPT}[k+1,j,b,mem] \end{array} \right\}$$

    Set $\text{aux1}[i,i+d,b,mem] \leftarrow k$, where k is the layer index minimizing $\texttt{OPTExact}[i,i+d,b,mem]$ in the above definition of OPTExact.
    Compute $\texttt{OPT}[i,i+d,b,mem]$ using the above definition of OPTfor each batch size b and available memory unit mem.
  end for
end for
$\texttt{OPT}[1,n,b,\text{TOT}]$ gives minimum inference time.

$$\begin{aligned} \texttt{OPT}[2,3,2,7] &= \texttt{OPTExact}[2,3,2,7] \\ &\qquad (b_1 = b = 2 \text{ in Equation } 2) \\ &= \texttt{OPT}[2,2,2,7] + \texttt{OPTExact}[3,3,2,7] \\ &\qquad + \texttt{OPT}[4,3,2,7] (k = 3 \text{ in Equation } 1) \\ &= \texttt{OPT}[2,2,2,7] + 3 + 0 \\ \texttt{OPT}[2,2,2,7] &= (2 * \texttt{OPTExact}[2,2,1,6])/2 \\ &\qquad (b_1 = 1, b = 2 \text{ in Equation } 2) = 4 \end{aligned}$$

Note that at each recursion all possible b_1 or ks are considered, so we are guaranteed to get those producing optimal result.

Practical Considerations for Edge Devices. To limit the number of DP table entries that are to be computed, we restrict the combinations of available memory to multiples of 2MB and 50 MB for the case of image and video processing respectively. This does not significantly affect the throughput gains.

Once the final OPT and OPTExact entries are computed, we only store the table entries that are actually used (in some optimal path). The size of this lookup table is much smaller than the DP table and the model size and thus does not contribute any significant overhead.

We emphasize that the computation of the dynamic program and setting up the lookup table are done offline and only done once. At inference time, only the appropriate optimal path needs to be queried from the table based on the available memory to figure out the batch sizes to be used for different layers. Thus our approach has just some additional table lookup overheads (over the uniform batch size inference) that do not add any significant time or energy consumption to the procedure.

The dynamic program can easily be extended to ensure that the latency of inferencing does not exceed some given bound. This can be achieved by modifying recurrence (1) so that whenever $\texttt{OPTExact}[\cdot,\cdot,\cdot,\cdot]$ exceeds the required latency threshold, we set it to ∞. This makes sure that our optimal solution never has larger latency.

Generalization to branched models: We now extend our dynamic programming algorithm to handle models comprising of branches. In such networks, there is a linear structure of layers as described previously. We call this the *main path*. Between two nodes of this main path, there can be multiple branches as illustrated in Figure 5 (a). This class of networks encompass more complex networks such as GoogleNet and ResNet. between two layers L_x and L_y, there may be multiple, say p, branches of layers $\langle \ell_{11},\ldots,\ell_{1n_1}\rangle$, …, $\langle \ell_{p1},\ldots,\ell_{pn_p}\rangle$.

Our dynamic program for handling this generic case works by separating out each of the layers between the two nodes L_x and L_y into an independent linear network; these are illustrated in Figure 5 (c). Then, all the branch layers appearing between L_x and L_y are collapsed (i.e., replaced) into a single layer called a *special* layer in the main path. This is illustrated in Figure 5 (b); here s denotes the collapsed layer. Note that there may be multiple such components in the network. Each one of these is collapsed separately and we get multiple special layers on the main path. This modification reduces the network to a simple linear structured network. Recall that in order to run our dynamic program on the main path, we require to initialize the table with entries of the form $\texttt{OPTExact}[i,i,b,mem]$ for each layer i and values of batch

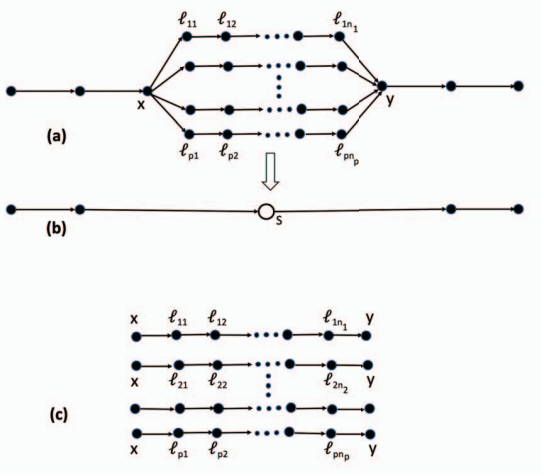


Figure 5. Handling special layer consisting of multiple branches.

size b and memory mem. This translates to computing the entries of the form $\texttt{OPTExact}[s, s, ., .]$ for a special layers L_s. We next describe how to compute these entries.

Let us focus on a particular special layer L_s, comprising of the initial layer L_x, the final layer L_y and p branches. For $\alpha = 1, 2, \cdots, p$, let $\ell_{\alpha 1}, \cdots \ell_{\alpha n_\alpha}$ be the layers in branch α (see Figure 9 (c)). Note that since each branch is itself a linear structure of layers, we can apply the dynamic program discussed in the previous section to each branch and obtain the DP table entries. In order to distinguish the DP entries of the branches from the main path, we use the notation $\texttt{OPTExact}[\cdot, \cdot, \cdot, \cdot]$ to refer to the table entries of the main path and $\texttt{OPTExact}_\alpha[\cdot, \cdot, \cdot, \cdot]$ to refer to the table entries of the α^{th} branch. Now, suppose we are computing $\texttt{OPTExact}(s, s, b, mem)$. Then the special layer s is being computed with a batch size of b. This implies that there are b samples at layer L_x. Each of the individual branches will process these b samples and produce b outputs at layer L_y. The computation for any branch α can be done with a batch size b or less, thus the branch α can process the b samples in multiple phases. Moreover, we assume that the branches are processed sequentially, i.e., branch $\alpha + 1$ will compute only after branch α finishes the computation for all the b samples. Note that each of the p branches comprising the special layer (say α^{th} branch) starts the processing with the input activation at layer L_x, passes it through the layers $\ell_{\alpha 1}, \ldots, \ell_{\alpha n_p}$ and finally aggregates the output at the output activation for layer L_y. Thus the input activations at layer L_x and output activations at layer L_y are required to be retained throughout the processing of the special layer L_s. Therefore the memory available for each of the branches to do the computation is $mem' = mem - \text{in}(x, b) - \text{out}(y, b)$, since the input and output activations of b samples need to be reserved at layers L_x and L_y respectively. Hence we get the following relation between the dynamic programming table entries for the main path and the component linear networks of the special layer L_s:

$$\texttt{OPTExact}(s, s, b, mem) = (1/b) \cdot \sum_{\alpha=1}^{p} \min_{1 \le b' \le b} \left\{ \begin{array}{l} b' \cdot \texttt{OPTExact}_\alpha(1, n_\alpha, b', mem') + \\ (b - b') \cdot \texttt{OPT}_\alpha(1, n_\alpha, b - b', mem') \end{array} \right\}$$

An intricacy in handling the branches is that for each branch α, the input activation size for samples of the first layer $\ell_{\alpha 1}$ and the output activation size for samples of the last layer $\ell_{\alpha n_\alpha}$ are taken to be zero. This is because this memory is already accounted for in the input to layer L_x and output of layer L_y.

IV. Experimental Results

We have conducted extensive experiments on state-of-the-art CNN models used in computer vision tasks. For video inference we have considered C3D network that extracts spatio-temporal features by employing 3D convolutions. Typically video models are computationally intensive and thus restrict the use of large batch sizes. For image inference we have considered four CNNs : GoogleNet, ResNet50, SqueezeNet-v0 and MobileNet-v1. C3D network is pretrained on UCF101 dataset and all image networks are pretrained on ImageNet-1K. Note that we do not modify the architecture or parameters of these networks and therefore their accuracy remains the same. Also, we consider variable batch sizes only during the inference phase and not during training these networks.

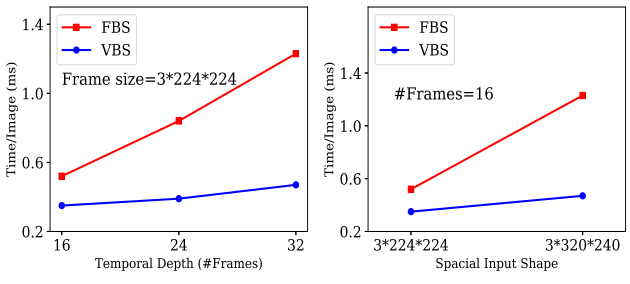
Baseline: We compare our VBS inference time with the baseline that uses uniform batch size across all the layers as prevalent in existing frameworks like Pytorch, TensorFlow etc. The batch size used by the baseline is determined by computing the maximum possible batch size that can be processed uniformly across all the layers under the given memory constraints. We refer to this baseline as Fixed Batch size (FBS).

For image inference, we consider a second baseline that can process different layers of the network using different batch sizes, however the batch size to process a particular layer is determined in a greedy fashion. For an input request to process a number of samples, say r samples, this baseline processes the samples sequentially along the layers of the network; thus layer i will start processing only after layer $i-1$ has completed processing all the r samples. A particular layer processes the r samples using the maximum possible batch size that can be employed under the memory constraints. Note that when layer i does the processing, we need to account for the input and output activation of all the r samples at layer i. We refer to this baseline as Greedy-VBS. This baseline is similar to the procedure presented in [23], where the authors consider breaking down the batch at a given layer into smaller batches in order to use more memory efficient cuDNN algorithms.

Implementation details: All experiments are conducted using Pytorch-0.4.1 framework with CUDA and cuDNN enabled. The inference time $\text{time}(i,b)$, input activation $\text{in}(i,b)$ and output activation $\text{out}(i,b)$ for each layer L_i and all batch sizes $b \in \{1,2,\cdots,12\}$ are computed by setting the eval() mode and torch.no_grad() flag in Pytorch. The working memory $\text{ws}(i,b)$ is obtained using the cuDNN debugging flag CUDNN_LOGINFO_DBG that provides information regarding the working space utilized by cuDNNConvolutionForward() calls for each layer. While calculating inference time and activation memory, we have bundled the ReLU, BatchNorm and Scaling layer with the preceding convolution layer since these layers are processed in-place. The experiments on video network were conducted on a single K80 GPU with a system memory of 12GB and the experiments on image networks were conducted on a simulated low resource environment where the memory was restricted during inference.

Performance gains: Our performance metric is the overall throughput gain, which is defined as the percentage reduction in per image inference time for our algorithm over the baseline.

We present two sets of results, the first set compares inference time of video network C3D with the baseline. Typically video networks with 3D convolutions are compute intensive and for a given model (here C3D) the maximum possible batch size depends on both the spatial dimension i.e., pixel size of each frame and temporal dimension i.e., number of frames in the input. We have demonstrated our results (see Figures 6a-6b) by varying both spatial and temporal dimensions. Figure 6a shows the comparison of inference time when the input spatial size is fixed at 3x224x224 and the temporal size is increased from 16 to 32. Usually video tasks demand larger temporal sizes in order to capture long-range time dependencies in the video activity [4]. Due to this we have varied the number of frames from 16 (minimum required by five Max Pooling layers with stride 2) to 32. Figure 6b shows the improvement in inference time with our algorithm when the number of frames is fixed (set to 16) and the spatial dimension is varied from 3x224x224 to 3x320x240 (increasing further the spatial dimension makes inference infeasible even for unit batch size for the system configuration used). These spatial dimensions are chosen corresponding to the input sizes of video datasets like UCF101 [26], HMDB51 [27], Sports1M [28] etc. In both the cases, our algorithm achieves throughput gain up to 61% that allows significant faster inference on video tasks like activity recognition, motion detection etc.



(a) Temporal (b) Spatial

Figure 6. Inference time for C3D network with varying input size

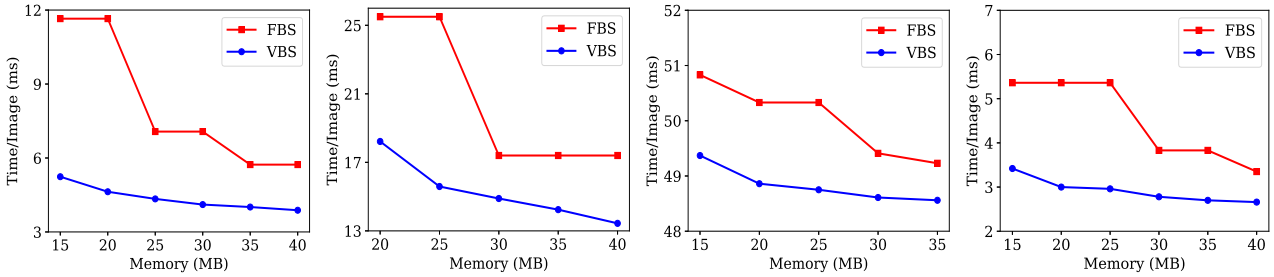
We note that our variable batch size approach demonstrates better performance gains over uniform batch size (FBS) approach in constrained memory settings where only small batch size can be executed. The reason goes back to the fact that the performance of core kernels like convolutions, matrix ops, worsens super-linearly as the problem size decreases (Figure 1). Thus an increase in batch size (in our approach) when the batch size is very small (in baseline) leads to significant gain in comparison to when the batch size is large (in baseline). This conforms to our observation in Figure 6 where the performance gains increase with increase in temporal/spatial depth. We observe that our gains are particularly higher under large input data size and constrained memory. For scenarios of using even larger temporal or spatial depth on GPUs with larger memory (than what considered in this paper), the batch size used (in baseline) will be very small, and our approach should give adequate gains.

Our second set of results compare the inference time of our approach with the baselines on state-of-the art image networks. Here, we study the throughput gains by simulating a low resource environment of edge devices like Omega2 [Omega2], Banana Pi-D1 [BananaPi] or Arduino boards [Arduino]. This is obtained by restricting the available working memory (in addition to the model size) between 15-40 MB. The minimum limit (15 MB) is set equal to the average memory required to process most of the CNNs with a batch of size one; the maximum limit (40 MB) is set equal to an estimated SRAM availability on the edge devices mentioned earlier.

Figure 7a- 7d shows the variation of the inference time (per image) with the available memory for our VBS algorithm and the FBS baseline. In all these experiments, the input request size is set to 12. Similar to the C3D inference scenario, our variable batch size approach provides better performance gains in constrained memory settings on image networks as well. In particular, we can consider two cases (i) available memory is very less (i.e., $<$ 30MB) and (ii) available memory is sufficiently large (i.e., $>$ 35MB). We

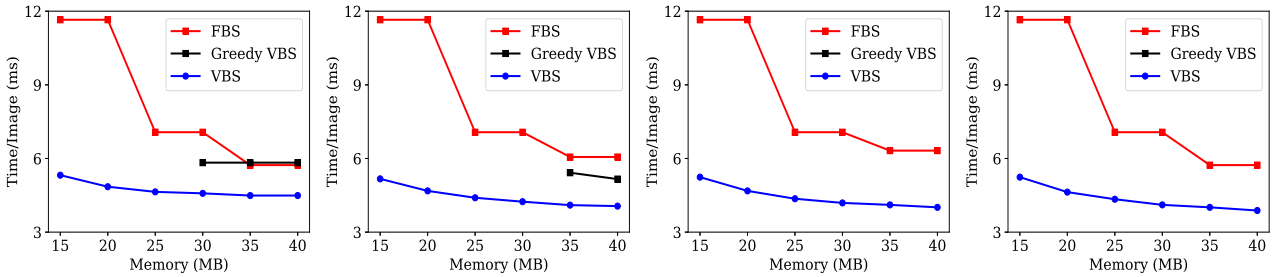
Network	Throughput gain (%)	
	Memory < 30 MB	Memory > 35 MB
GoogleNet	60.25	36.55
ResNet50	38.91	22.75
SqueezeNet-v0	44.03	29.50
MobileNet-v1	5.68	1.61

Table II
THROUGHPUT GAIN (MAX) FOR NETWORKS WITH VARYING MEMORY



(a) GoogleNet (b) ResNet50 (c) MobileNet-v1 (d) SqueezeNet-v0

Figure 7. Inference time comparison for a) GoogleNet, b) ResNet50, c) MobileNet-v1 d) SqueezeNet-v0 with varying available memory. As the available memory increases the batch sizes processed by the model also increases. For eg. maximum batch size (that can be processed by any layer) in case of ResNet50 with 25MB and 30MB memory sizes are 6 and 8 respectively. At these memory values the baseline is able to do only 1 and 2 batch sizes respectively. Detailed batch sizes used by different layers for GoogleNet at 20MB available memory can be observed in Figure 9



(a) Input Request Size = 6 (b) Input Request Size = 8 (c) Input Request Size = 10 (d) Input Request Size = 12

Figure 8. Inference time comparison for a) GoogleNet, b) ResNet50, c) MobileNet-v1 d) SqueezeNet-v0 with varying available memory. As the available memory increases the batch sizes processed by the model also increases. For eg. maximum batch size (that can be processed by any layer) in case of ResNet50 with 25MB and 30MB memory sizes are 6 and 8 respectively. At these memory values the baseline is able to do only 1 and 2 batch sizes respectively. Detailed batch sizes used by different layers for GoogleNet at 20MB available memory can be observed in Figure 9

observe that when the available memory is less than 30 MB, the baseline is restricted to work with smaller batch sizes (say b). On the other hand, our algorithm optimizes the batch sizes for each layer by using low ($<b$) batch sizes for some and high ($>b$) batch sizes for other layers thereby resulting in faster prediction time. With sufficiently large available memory (>35MB), the baseline can process large batch sizes, and hence its performance approaches our algorithm. Our approach results in throughput gains up to 60% on GoogleNet for memory < 30 MB and up to 36% for memory > 35 MB (refer Table II for throughput gains for different networks and varying memory sizes). We also observe that the throughput gains on MobileNet-v1 are limited to 6%. This is because most convolution layers of MobileNet-v1 require similar computational memory (std. deviation in computational memory is just 1.45 MB) during forward pass, whereas our approach works best when there is significant variation among layers.

Figure 8a- 8d compares the inference time (per image) for our VBS algorithm with both the FBS and the Greedy-VBS baselines for the GoogleNet network for different input request sizes (varied from 6 to 12). We observe that the Greedy-VBS performs better than the FBS when available memory is sufficiently large, or when the input request size is small. This is because the Greedy-VBS approach can process some layers (with smaller memory requirements) with larger batch sizes than that used by the FBS, leading to performance gains over the FBS. However, as stated before, the Greedy-VBS approach needs to store the input and output activation of all the requested samples at a particular layer during processing of that layer. Hence for smaller memory ranges (<30MB when input request size is 6 and <35MB when input request size is 8) or larger input request sizes (10 and 12), the Greedy-VBS approach becomes practically infeasible. Our variable batch size approach outperforms both the baselines over all input request sizes and memory range; the gains are particularly higher in constrained memory settings as explained above.

Batch size distribution across layers. We next analyze how our dynamic program selects different batch sizes for different layers. Figure 9 shows the batch sizes selected by our algorithm for a single inception module of GoogleNet when the available memory is 20 MB. The numbers beside the layer names in the figure show the batch sizes selected by

our algorithm for that layer in different phases. For example, the layer "inception_5b/output" can process 8 samples at a time. The input for this layer is accumulated over two phases of processing of previous layers like "inception_3a/output", which executes 6 samples in the first phase and 2 in the second. On the other hand, the layer "inception_3a/3x3" requires more memory for computation, and can't afford to process 6 samples in a single phase; rather it processes those 6 sample in two phases of 3 samples each. In its third phase, the layer "inception_3a/3x3" processes 2 images. The baseline is able to process only 2 images across every layer in each phase. These observations conform to our hypothesis that layers requiring large memory for input/output activations should be computed with smaller batch sizes and vice-versa.

Memory utilization across layers: Finally, we analyze how our VBS algorithm makes better utilization of the memory in order to achieve better throughput. Table III compares the layer-wise un-utilized memory for the baseline with our approach, under the same setting (available memory=20 MB). It is clear that our approach is able to make better use of memory by employing different batch sizes for different layers. We observe that the free memory is mostly below 2MB. The baseline, on the other hand, is restricted to use a smaller batch size (constrained by the maximum memory consuming layer of the network), and use of this small batch size across all the layers leads to inefficient memory utilization for most layers. We notice that for the baseline the

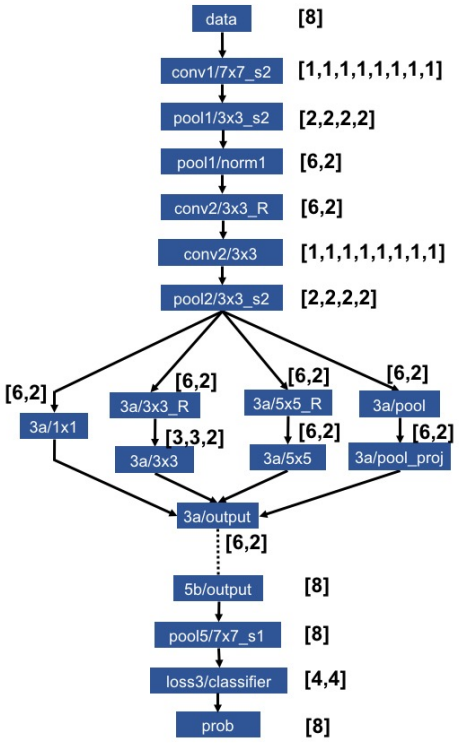


Figure 9. Batch sizes used by GoogleNet for 20 MB memory.

Layer Name	Un-utilized memory (MB)	
	Baseline	VBS Algo.
conv1/7x7_s2	9.29	0.30
pool1/3x3_s2	12.99	0.00
pool1/norm1	16.93	0.38
conv2/3x3_reduce	17.69	0.13
conv2/3x3	9.63	0.38
pool2/3x3_s2	14.67	1.33
inception_3a/output	18.66	1.96
inception_5b/output	19.65	6.9
pool5/7x7_s1	19.61	6.91
loss3/classifier	16.08	1.67

Table III
UN-UTILIZED MEMORY WHILE PROCESSING INDIVIDUAL LAYERS OF GOOGLENET WITH 20 MB AVAILABLE MEMORY. BATCH SIZES USED BY OUR APPROACH AT THE DIFFERENT LAYERS FOR THIS SETTING IS PRESENTED IN FIGURE 9.

un-utilized memory is more than 15MB for many layers, thus most layers are hardly able to utilize the available memory of 20MB.

V. CONCLUSIONS AND FUTURE WORK

In this paper we propose an alternative paradigm where instead of processing every layer with the same batch, we use different batch sizes for different layers depending on their computational requirement. We propose a dynamic program based algorithm to determine the optimal batch sizes to be used for each layer in order to maximize the inference throughput. We demonstrate that our algorithm can improve the inference throughput up to 60% for state-of-the-art video and image CNN models. Our DP can easily be extended to optimize the battery/energy consumption; this can simply be done by filling the dynamic table entries in the base case with battery/energy consumption instead of time. Our approach can also be applied for data distributed inference, since here each node can independently apply the dynamic program for the optimization. An interesting direction for future work is to use variable batch sizing to speed up the training process.

Our proposed method improves throughput for most state-of-the-art DNNs for image classification, video inference etc, where the activation sizes (and/or the working memory) vary with layers. This concept can be extended to networks used in other domains as well, like image segmentation (for example, FCN-16s, SegNet, RefineNet etc that use encoder-decoder architecture).

REFERENCES

[1] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI, 2017, USA.*, 2017, pp. 4278–4284.

[2] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *CVPR, USA*, 2016, pp. 779–788.

[3] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE*

Trans. Pattern Anal. Mach. Intell., vol. 39, no. 4, pp. 640–651, 2017.

[4] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *ICCV*, 2015, pp. 4489–4497.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[6] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *JETC*, vol. 13, no. 3, pp. 32:1–32:18, 2017. [Online]. Available: http://doi.acm.org/10.1145/3005348

[7] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016. [Online]. Available: http://arxiv.org/abs/1611.06440

[8] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[9] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711

[10] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016. [Online]. Available: http://arxiv.org/abs/1612.01064

[11] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *CoRR*, vol. abs/1412.6115, 2014. [Online]. Available: http://arxiv.org/abs/1412.6115

[12] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, 2016, pp. 525–542.

[13] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[14] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14, 2014, pp. 1269–1277.

[15] Y. Ioannou, D. P. Robertson, R. Cipolla, and A. Criminisi, "Deep roots: Improving CNN efficiency with hierarchical filter groups," in *CVPR 2017, USA*, 2017, pp. 5977–5986.

[16] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *CoRR*, vol. abs/1511.06530, 2015. [Online]. Available: http://arxiv.org/abs/1511.06530

[17] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.

[18] A. RoyChowdhury, P. Sharma, E. Learned-Miller, and A. Roy, "Reducing duplicate filters in deep neural networks," in *NIPS workshop on Deep Learning: Bridging Theory and Practice*, 2017.

[19] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016.

[20] 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," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[21] T. Starner, "Project glass: An extension of the self," *IEEE Pervasive Computing*, vol. 12, no. 2, pp. 14–16, April 2013.

[22] S. Lym, A. Behroozi, W. Wen, G. Li, Y. Kwon, and M. Erez, "Mini-batch serialization: Cnn training with inter-layer data reuse," *arXiv preprint arXiv:1810.00307*, 2018.

[23] Y. Oyama, T. Ben-Nun, T. Hoefler, and S. Matsuoka, "Accelerating deep learning frameworks with micro-batches," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 402–412.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[26] K. Soomro, A. R. Zamir, M. Shah, K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *CoRR*, 2012.

[27] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, "HMDB: a large video database for human motion recognition," in *ICCV*, 2011.

[28] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *CVPR*, 2014.

[29] Omega2, https://onion.io/omega2/.

[30] BananaPi, http://www.banana-pi.org/d1.html.

[31] Arduino, https://www.arduino.cc/.