TASO: Time and Space Optimization for Memory-Constrained DNN Inference

YUAN WEN, Trinity College Dublin
ANDREW ANDERSON, Trinity College Dublin
VALENTIN RADU, University of Edinburgh
MICHAEL O'BOYLE, University of Edinburgh
DAVID GREGG, Trinity College Dublin

Convolutional neural networks (CNNs) are used in many embedded applications, from industrial robotics and automation systems to biometric identification on mobile devices. State-of-the-art classification is typically achieved by large networks, which are prohibitively expensive to run on mobile and embedded devices with tightly constrained memory and energy budgets. We propose an approach for ahead-of-time domain-specific optimization of CNN models, based on an integer linear programming (ILP) for selecting primitive operations to implement convolutional layers. We optimize the trade-off between execution time and memory consumption by: 1) attempting to minimize execution time across the whole network by selecting data layouts and primitive operations to implement each layer; and 2) allocating an appropriate workspace that reflects the upper bound of memory footprint per layer. These two optimization strategies can be used to run any CNN on any platform with a C compiler. Our evaluation with a range of popular ImageNet neural architectures (GoogleNet, AlexNet, VGG, ResNet and SqueezeNet) on the ARM Cortex-A15 yields speedups of 8× compared to a greedy algorithm based primitive selection, reduces memory requirement by 2.2× while sacrificing only 15% of inference time compared to a solver that considers inference time only. In addition, our optimization approach exposes a range of optimal points for different configurations across the Pareto frontier of memory and latency trade-off, which can be used under arbitrary system constraints.

Additional Key Words and Phrases: neural network optimization, computing operators, primitive selection, optimal convolutional layer, memory-time trade off

ACM Reference Format:
Yuan Wen, Andrew Anderson, Valentin Radu, Michael O'Boyle, and David Gregg. 2020. TASO: Time and Space Optimization for Memory-Constrained DNN Inference. 1, 1 (May 2020), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

CNNs are increasingly deployed in embedded devices for applications such as: vision based perception and control [16], autonomous cars [4]; in the wearable technology and medical space with wearable cameras assisting visually impaired [22], to rehabilitate memory loss [9], and to diagnose cancer with human-level efficiency consistently [7].

These emerging computer applications are deployed on mobile and embedded devices, which interact with real world data such as video and audio. With limited compute resources, large neural network models will need to be specially optimized to meet the constraints of each device. CNNs are not only specific to computer vision, but also find applications in speech recognition [23] and natural language processing [13, 14].

However, CNN models often approach hundreds of megabytes (ResNet-50 is about 150 megabytes). Significant research effort has been invested in reducing model size, with MobileNet [10] and SqueezeNet [11] being prominent

Authors' addresses: Yuan Wen, weny@tcd.ie, Trinity College Dublin; Andrew Anderson, andersan@tcd.ie, Trinity College Dublin; Valentin Radu, vradu@inf.ed.ac.uk, University of Edinburgh; Michael O'Boyle, mob@inf.ed.ac.uk, University of Edinburgh; David Gregg, david.gregg@cs.tcd.ie, Trinity College Dublin.

2020. Manuscript submitted to ACM

examples of smaller models. However, even reduced-sized models often require too much processing and memory capacity that is available on resource-constrained embedded systems. The alternative is to drastically improve the computational and memory efficiency of larger models to meet the hardware constraints on target platforms – this is the focus of our work.

In this paper, we address the problem of optimizing the execution of CNNs under tight inference time and memory footprint budgets. Many different implementations exist for performing the operations of convolutional layers, some sacrificing memory for performance (*e.g.*, *im2col*), while others applying radical transformations to benefit from cheaper operations (*e.g.*, *winograd*). Further, the execution time of an algorithm depends partly upon the layout of data in memory. For example, a common data layout for inputs is CHW (channels, height width), but some algorithms are faster on a HWC data layout.

Using a library of more than 57 primitives implementing convolution via different algorithms and data layouts, we evaluate the optimal primitive selection for each convolutional layer with respect to the overall execution time of the network, and the memory footprint. While other solvers have been used for optimising the primitive selection for inference time ($e.g.$, PBQP [1]), on embedded systems memory constraint is more important and has not been considered for primitive selection before. We explore two optimization strategies: 1) focused on enhancing neural network performance if input, output and working data for all layers of the neural network can fit into main memory; and 2) introducing a *workspace* which is reused by each layer at a time, with the size determined by the ILP optimizer. This ensures that neural networks can be executed on a device with very small memory size, enough to accommodate the requirements of at least on primitive of a convolutional layer, which otherwise would have been impossible to run via a default memory-hungry primitive ($e.g.$ im2col).

This paper makes the following contributions:

- We develop an ILP formulation which can expose the trade-off between execution time and memory footprint of CNNs with varying primitives across convolutional layers.
- We define two optimization strategies for the ILP optimizer, one enhancing the performance across the whole network, while the second imposing a memory budget per layer, which can be applied to whole network in memory and layer by layer in memory respectively.
- We evaluate our approach using 5 popular CNN (GoogleNet, VGG, AlexNet, SqueezeNet and ResNet) on an edge processor (ARM Cortex-A15).

The remainder of this paper proceeds as follows. Motivation for this work is presented in Section 2. In Section 3 we give some background on how convolutional layers in CNNs work. Section 4 presents our methodology for selecting primitives, including the full ILP formulation, and this is evaluated in Section 5, followed by comparison with other solvers. This paper ends with related work (Section 6) and conclusions (Section 7).

2 MOTIVATION

Intelligent systems are becoming pervasive at an accelerated rate [20], although many of these systems still rely on sending data to the cloud to perform inferences with heavy machine learning models. However, this approach is under increasing scrutiny with pressure to perform more of these computations on device. Sending data to the cloud requires large amounts of energy for data transmission, and creates privacy concerns for the users of these devices.

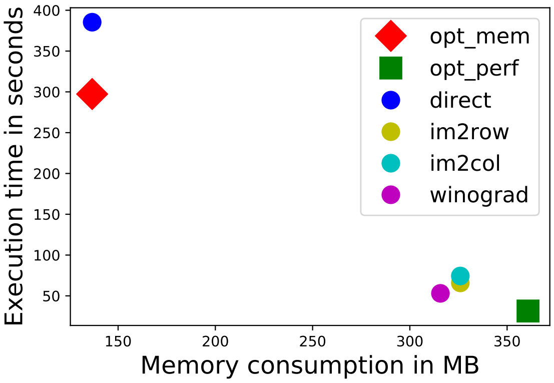


Fig. 1. Execution time and memory size of GoogleNet on an ARM A15 processor, with the same primitive choice used across the entire network. We also present the optimal memory (*opt_mem*) and optimal time performance (*opt_perf*) with primitives selection from across our full range of 57 primitives.

2.1 Intelligence Migrating to Edge Computing Systems

There is a growing interest to port machine learning to edge computing devices due to a growing suite of applications.

Another reason for migrating computations to the edge is to operate without Internet connectivity even in remote locations, whiles also being independent of connection quality (critical for some applications like self-driving cars). It is thus expected that more deep neural network models will need to be run on devices with tight computation and memory constraints.

2.2 Running Large Models on Constrained Hardware

Current deep neural network models are mostly designed for high performance server-side computations where memory constraints are more relaxed. Adopting these models on smaller devices is not straightforward. Figure 1 shows the run-time memory requirement of GoogleNet (fitting the whole model into memory), implementing the convolutional layer with different computing primitive choices. One option is to implement the network with a single primitive choice across all convolutional layers (as commonly done by most deep learning frameworks): direct convolution (*direct*), image to row (*im2row*), image to column (*im2col*) and winograd; and the alternative is to select primitives at each convolutional layer to optimize for memory footprint (*opt_mem*) and for inference time performance [1] (*opt_perf*). One choice of primitives across the whole network may result in very poor performance (direct convolution running one inference of GoogleNet in 400sec) or may not be able to run the network at all, if physical memory on device is less than 300MB. In contrast, optimizing primitive selection and data layouts for each layer enables more control, not just better execution time (*opt_perf*) and minimal run-time memory footprint (*opt_mem*), but also additional optimal points in between to adjust the trade-off for device and application conditions.

Another approach to enable large models on smaller devices is through compression, quantization and pruning of models [8]. However, these require extensive effort from data scientists to synthesize new models from original large ones and despite their effort these still achieve lower classification accuracy than original models. To reduce migration cost and to retain the intended inference quality, we consider only methods that produce the same convolutional operation, albeit implemented with different primitives that produce the intended output.

Fig. 2. Memory footprint at run-time of largest layers in some of the most popular deep neural networks.

2.3 Convolutional Layer Workspace

On many devices, memory is very limited (tens of MB) so loading the entire network is not a viable approach. In this situation, computations can be performed layer by layer, loading only the weights necessary for one layer at a time. However, each convolutional primitive has different memory requirements. Figure 2 shows that one order of magnitude difference exists between the memory requirement of performing the convolution with direct versus using the *im2col* approach. This difference in the amount of memory required by each primitive should instruct the primitive selection to: a) make sure that a layer can fit into memory b) exploit the available space in memory to choose a better performing primitive within the memory budget. This memory budget is what builds our workspace, which is presented in the following sections.

3 BACKGROUND

Convolutional neural networks consist of a graph of *layers*. Each layer is a standard component such as convolution layers, pooling layers, and fully-connected layers. The great majority of execution time in CNNs is spent in convolution layers [12]. Convolution layers in CNNs operate somewhat differently in traditional convolution in image processing. Rather than applying a single convolution kernel to a single image, CNNs convolve an inputs with many channels (typically dozens or hundreds) with many-channel kernels. Further, in each convolution layer in the input is convolved with many (typically 32-1024) different multichannel kernels. The result is that convolution layers operate on large input and kernel tensors, and are computationally very intensive.

There are a number of different algorithms for computing CNN convolution, each of which has its own strengths and weaknesses. One the earliest successful approaches, *im2col*, lowers the input (image) into a column matrix, and then computes the multiple-channel multiple-kernel convolution using matrix multiplication. This allows the use of highly-optimized matrix multiplication libraries, which are available for most processors. However, for a $K \times K$ convolution, lowering increases the size of the input by a factor of K^2. In contrast, the Memory-Efficient Convolution (MEC) algorithm performs K separate matrix multiplications, and increases the input size by a factor of K [5]. The GEMM-accumulating *kern2row* algorithm requires a only a sub-linear increase in space, and performs K^2 matrix multiplications [2]. Another approach is to implement convolution as a simple loop nest, but this loses the benefit of being able to leverage high-performance matrix multiplication libraries.

Fast convolution algorithms rely on mathematical identities to reduce the number of operations needed. By far the best known fast algorithm is FFT convolution, which reduces the complexity of convolving two signal of length n from $O(n^2)$ to $O(nlog(n))$. However, the dimensions of the convolution kernels in CNNs are almost always very small, typically 3 or 5×5, and algorithms that are specifically designed for short, real-valued convolutions typically require fewer operations. Winograd convolution [15] greatly reduces the operation count for CNN convolution, at the cost of linear transforms of the input and output which require additional computation and significant memory.

CNN frameworks such as TensorFlow, Pytorch and Caffe implement convolution layers by calling out to specialized libraries that implement these various convolution algorithms.

4 PRIMITIVE SELECTION WITH INTEGER LINEAR PROGRAMMING

We model the task of configuring a DNN to achieve the best inference time under memory constraint as an integer linear programming (ILP) problem. Typical DNNs can be viewed as a graph of layers, where convolution layers dominate the execution time, and often consume a significant amount of memory. Each convolutional layer in the graph can be implemented by one of a large set of candidate *primitives*, each with their own execution time (*cost*), input and output data transformation costs, and memory requirement.

In prior work, Partitioned Boolean Quadratic Programming (PBQP) was shown to be efficient in minimizing the execution time of convolutiona neural networks by selecting the optimal primitives for each layer [1]. However, that earlier work does not model memory requirements, and cannot set a maximum memory budget. This aspect is crucial to embedded devices with limited memory. Here, we propose a novel approach that optimizes for both memory budget and execution time, which is a much harder problem. Unlike previous work, our control over memory requirement allows fine adjustments of the time-memory trade-off to generate the optimal network for any application requirements.

4.1 Integer Linear Programming

Linear programming is a mathematical model that seeks to minimize an objective function subject to linear constraints. In *Integer* Linear Programming, all decision variables are constrained to integers and hold a linear relationship in the model. ILP solver searches for a solution that meets the constraints of all the linear relationships, while seeking to minimize the objective function. We use the formulation in [19] to define this optimisation:

$$\begin{array}{rlcll} \min\limits_{x} & c^T x & & & \\ \text{s.t.} & Ax & \leq & b & \\ & x & \geq & 0, & x \in \mathbb{Z}^n \end{array} \tag{1}$$

where x represents decision integer variables to be determined, c, b are coefficient vectors, A is a coefficient matrix, and $Ax \leq b$ and $x \geq 0$ are the constraints. We search for the solution x, an integer sequence which satisfies the given constraints with the lowest cost objective function.

4.2 ILP Model

Equations 2 and 3 show how our task can be optimised for two objectives, for DNN execution time and for memory requirement, respectively. In this formulation, decision variable x is a vector of primitives selected to implement each layer of the DNN. Objective functions $c^T_{execution}x$ and $c^T_{memory}x$ are the *accumulated* execution time and memory usage, respectively. Whereas restrictions $A_{mem}x \leq M$ and $A_{exec}x \leq E$ are user-supplied constraints on total execution time (E) and memory (M) budgets for an inference with the configured DNN.

ILP finds the optimal solutions, one or multiple solutions if these exist, or no acceptable solution if one does not exist. We choose the first determined solution when there are more than one solutions. If the is no memory or inference time budget provided (no imposed constraints) then ILP finds the global optimal x under the objective function. In that situation, ILP searches for the primitive configuration yielding either best performance (using $c^T_{execution}x$, the same as using PBQP [1]) or least memory consumption (using $c^T_{memory}x$).

$$\begin{array}{rlcll} \min\limits_{x} & c^T_{execution}x & & & \\ \text{s.t.} & A_{mem}x & \leq & M & \\ & x & \geq & 0, & x \in \mathbb{Z}^n \end{array} \tag{2}$$

$$\begin{array}{rlcll} \min\limits_{x} & c^T_{memory}x & & & \\ \text{s.t.} & A_{exec}x & \leq & E & \\ & x & \geq & 0, & x \in \mathbb{Z}^n \end{array} \tag{3}$$

4.3 Component Costs

The overall costs of a solution comprising total execution time and total memory footprint is composed of several component costs.

The execution time consists of primitive execution time itself and the time spent on data layout transformations of input tensor. Note that we do not use a single canonical data layout for the input, output and kernel tensors used by the convolution primitive. Instead, each primitive specifies the data layouts for input and output, such as *CHW* or *HWC* (*C* being the number of channels, *H* and *W* input height and width respectively). These layouts have a different performance on a devices under various convolution algorithms, consuming a specific amounts of memory based on the layer configuration. In addition, if the output format of a chosen primitive does not match the input format of that layer, our framework has to insert a data layout transformation routine to convert the output layout. The execution time and memory cost of these data layout transformations are also considered in our solver.

Formula 4 presents this execution time for a layer comprised of primitive execution time and data layout transformation time. If the input format is the same as the required output format then data transformation is not needed, associating a zero cost. On the memory side, there is no requirement for extra memory to perform these transformations, since the input and output buffers can be used in place.

The objective functions are expanded in formulas 4, 5, 6 and 7 for a network with m convolutional layers, where x_i is the selected primitive for the corresponding layer. In formula 6, the matrix $Trans\_Mat_{i\_(i+1)}$ is a transformation matrix that contains the profiled cost of transforming a tensor data layout between neighbouring convolution layers i and $i + 1$ given the choice of algorithms selected to implement layers i and $i + 1$.

$$Time = T\_algm\_exe + T\_format\_trans \tag{4}$$

$$T\_algm\_exe = \sum_{i=1}^{m} x_i \times t_i \tag{5}$$

$$T\_format\_trans = \sum_{i=1}^{m-1} Trans_{i\_(i+1)}[x_i, x_{i+1}] \tag{6}$$

$$Memory = \sum_{i=1}^{m} x_i \times m_i \tag{7}$$

Figure 3 shows the data layout transformation matrix. Assuming there are n primitive implementations for both layer i and following layer j. The $Trans\_Mat$ is therefore an n-by-n matrix. Each row of the matrix represents the primitive used on layer i, and columns present the possible primitives for layer j. Each value on this matrix represents the data layout transformation cost for going from the output format of the first layer to the input format of its following layer. We measure these values during device profiling.

For any given layer, exactly one primitive can be selected to perform the convolution. Therefore, for each layer, the decision variable x_i is a vector of size n with each element having a value of either 0 (primitive not selected) or 1 (primitive selected), with n being the number of candidate primitives. Since only one primitive will be selected per layer, the vector of decision variables forms a one-hot vector.

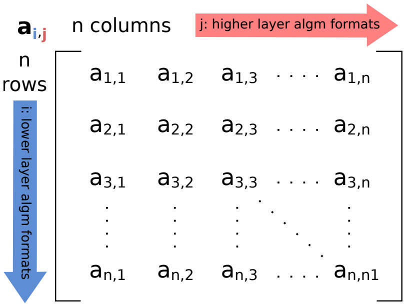

a i,j
n columns
j: higher layer algm formats
n rows
i: lower layer algm formats
a1,1 a1,2 a1,3 . . . . a1,n
a2,1 a2,2 a2,3 . . . . a2,n
a3,1 a3,2 a3,3 . . . . a3,n
an,1 an,2 an,3 . . . . an,n1

Fig. 3. Matrix contains the cost for data layout transformation between neighbouring layers.

Equations 8 and 9 show the calculation of primitive execution time and memory footprint, where $\vec{L_i}$ represents the one-hot condition variable vector for layer i, $\vec{T_i}$ is the profiled execution time vector for all candidate primitives for a layer, and $\vec{M_i}$ contains the corresponding memory footprint for each primitive.

$$T\_algm\_exe = \sum_{i=1}^{m} \vec{L_i} \bullet \vec{T_i} \tag{8}$$

$$Memory = \sum_{i=1}^{m} \vec{L_i} \bullet \vec{M_i} \tag{9}$$

Similarly, we can model the time required to perform data layout transformations in the same way as above, obtaining equation 10.

$$T\_format\_trans = \sum_{i=1}^{m-1} \vec{L_i} \times Trans\_i - (i+1) \times \vec{L_{i+1}}^{T} \tag{10}$$

To notice here is that equation 10 is not strictly a linear problem due to the multiplication between every two nodes. However, the decision variable vectors are one-hot vectors, which allows us to transform this equation back to a linear form by evaluating the *floor* function of the sum of the decision variables.

$$Trans_(i, (i+1)) = \sum_{j=1}^{n} \sum_{k=1}^{n} \lfloor (L_{i_j} + L_{(i+1)_k})/2 \rfloor \times Trans(i, (i+1)) \tag{11}$$

$$Time = \sum_{n=1}^{m} \vec{L_n} \bullet \vec{T_n} \quad + \quad \sum_{i=1}^{m-1} \sum_{j=2}^{m} \vec{L_i} \times Trans\_(i, j) \times \vec{L_j}^{T} \tag{12}$$

4.4 Whole network performance optimization with ILP

This optimization strategy is designed for devices with enough physical memory to accommodate the entire network into main memory, optimizing for performance across the whole network. A memory budget can be introduced if sharing the system memory with other tasks and selecting performance critical conditions to be met. The ILP optimiser finds the optimal primitive selection on each layer of a network to meet the imposed constraints.

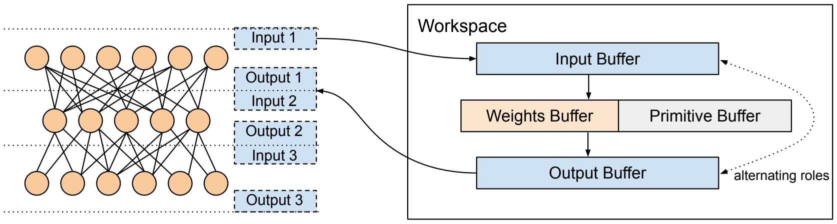


Fig. 4. Reusing three allocated buffers (input, output and weights) by flipping the roles of the input and output buffer after each convolutional layer, with the output of one layer becoming the input of following layer.

$$
\begin{aligned}
&\mathit{minimize} \\
&\mathit{Time} = \sum_{i=1}^{m} \vec{L_i} \bullet \vec{T_i} \quad + \quad \sum_{i=1}^{m-1}\sum_{j=2}^{m} \vec{L_i} \times Trans\_(i,j) \times \vec{L_j}^{T} \\
\text{s.t.} \quad & \sum_{i=1}^{m} \vec{L_i} \bullet \vec{M_i} \quad \leq MemoryBudget \\
& \vec{L_i} = \{x_j | j \in i \times n \; to \; (i+1) \times n \quad \wedge \quad n = num \; of \; primitives \quad \wedge \quad x_j = 0,1\} \\
& \sum_{j=1}^{n} \vec{L_{i_j}} = 1, \quad \forall i \in [1,m]
\end{aligned}
\tag{13}
$$

Formula 13 optimizes for inference time under constrained imposed by the allowed memory consumption, while Formula 14 optimizes memory footprint with constrains from execution time. Varying these constrains allows us to find a Pareto of optimal points under imposed memory or performance constraints as presented in the evaluation section.

$$
\begin{aligned}
&\mathit{minimize} \\
&\mathit{Memory} = \sum_{i=1}^{m} \vec{L_i} \bullet \vec{M_i} \\
\text{s.t.} \quad & \sum_{i=1}^{m} \vec{L_i} \bullet \vec{T_i} \quad + \quad \sum_{i=1}^{m-1}\sum_{j=2}^{m} \vec{L_i} \times Trans\_(i,j) \times \vec{L_j}^{T} \quad \leq PerformanceBudget \\
& \vec{L_i} = \{x_j | j \in i \times n \; to \; (i+1) \times n \quad \wedge \quad n = num \; of \; primitives \quad \wedge \quad x_j = 0,1\} \\
& \sum_{j=1}^{n} \vec{L_{i_j}} = 1, \quad \forall i \in [1,m]
\end{aligned}
\tag{14}
$$

4.5 Workspace-based optimization (layer by layer)

If device memory is limited such that the entire neural network does not fit into main memory with any combination of primitives, we consider the alternative optimization strategy, layer by layer. This is solved using the ILP solver also. A workspace size is identified to represent the upper bound of memory footprint to be used by primitives for each layer in the network, one at a time.

Figure 4 presents the component buffers of the workspace. This includes two alternating input-output buffers, a buffer to hold the layer weights. The weights of each layer of the network are loaded into the workspace in progression. Additional buffers hold transformations and intermediary values.

The size of this workspace is determined by equation 15 below.

$$
\begin{aligned}
&\quad minimize \\
&|Workspace| = \max_{i=1}^{m} \vec{L_i} \bullet \vec{M_i} \\
&\text{s.t.} \quad \sum_{i=1}^{m} \vec{L_i} \bullet \vec{T_i} \quad + \quad \sum_{i=1}^{m-1} \sum_{j=2}^{m} \vec{L_i} \times Trans\_(i,j) \times \vec{L_j}^{T} \quad \leq PerformanceBudget \\
&\qquad \vec{L_i} = \{x_j | j \in i \times n\ to\ (i+1) \times n \quad \wedge \quad n = num\ of\ primitives \quad \wedge \quad x_j = 0,1\} \\
&\qquad \sum_{j=1}^{n} \vec{L_{i_j}} = 1, \quad \forall i \in [1,m]
\end{aligned}
\tag{15}
$$

This method follows a first stage of performance optimization across the whole network as indicated by equation 13 to identify the appropriate workspace size. Equation 15 differs from equation 14 by performing an optimization over the maximum memory required across all layers, as opposed to accumulating memory when all network layers are simultaneously in memory.

5 EVALUATION

To evaluate our approach we implemented the ILP solver and evaluate it in this section on selecting primitive routines for convolutional layers to run on a low power embedded processor core. We measure the execution time and memory requirement for each convolutional layer implemented with one of the 57 convolution primitive routines in the profiling stage. These device profile measurements are the input for the ILP solver, which selects the optimal primitive for each layer.

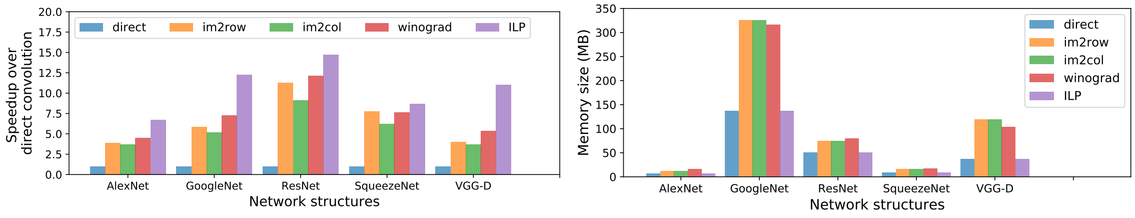
5.1 Experimental Setup

We use the ODroid-XU4 embedded development board, which includes the Samsung Exynos-5422 processor. The Exynos-5422 has the low-power embedded core 2.1GHz ARM Cortex-A15. The Cortex-A15 is a powerful 32-bit out-of-order superscalar core, with NEON instructions. We concentrate on the CPU because this is the most common computation unit found in embedded devices, although where available, the GPU can be utilized similarly.

Our ILP solver based approach is not tied to any specific CNN architecture or hardware platform, being generic to determine the best primitive selection solution for any CNN and environment. This is implemented with the open-source PuLP python package.

This evaluation is performed on five popular deep learning models (AlexNet, VGG, GoogLeNet, ResNet and SqueezeNet) due to: (1) they are highly accurate for image recognition tasks, as shown by their success in the ILSVRC 2014 competition; (2) they have a relatively complex network structure, with forks and joins, rather than a simple sequence of layers.

Our results present only the inference process (not the training phase) of an already trained neural network. The chosen evaluation scenario reassembles a streaming application, processing a single frames at a time.



(a) Performance speedup over all direct convolution.

(b) Memory footprint.

Fig. 5. Performance and memory optimisations with ILP for five popular CNNs on the ARM Cortex 15.

Table 1. The time required by our solver to find a solution for each CNN, using an Intel Core i5 CPU. This is after each layer has been profiled on the target device in the offline phase.

Network	Number of conditional variables	Time (in sec)
AlexNet	456	3.373
VGG	741	5.259
SqueezeNet	1710	8.835
GoogleNet	3990	23.84

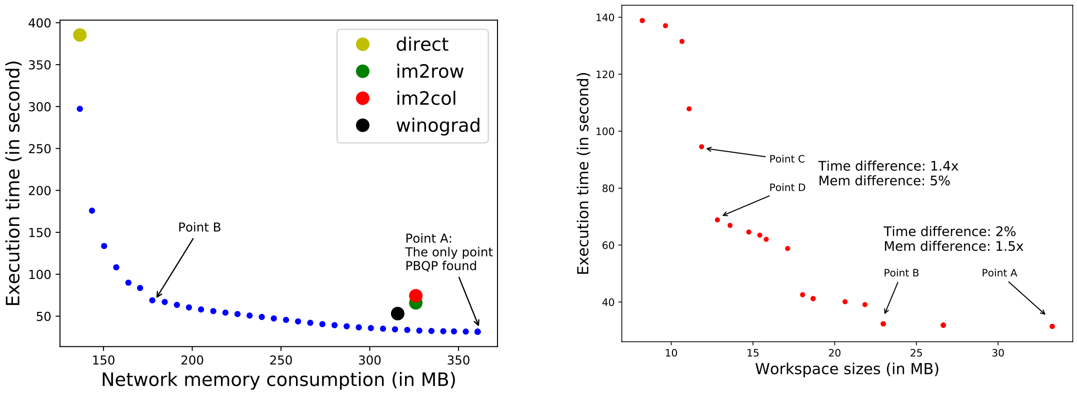
5.2 ILP and Constraints

The task in to select the optimal primitive routine for each convolutional layer within a CNN. This is done using the profiled execution time of each primitives on the target device (ODroid-XU4). We optimise the selection with our ILP solver, with the constraint of a given memory budget and optimising the execution time. This generates a Pareto optimal frontier of best primitive selection solutions for varying memory budget. In this experiment, we optimize for best performance over the entire network (whole network in memory).

To find the optimal solution, the ILP solver takes 23.84 seconds on the layers of GoogleNet. Table 1 presents such cost for all networks we discussed in this paper. Given that configured networks generally work on devices for a long period of time (millions or billions of inferences), the one-time cost of the ahead-of-time optimization can be considered trivial.

Figure 5a shows the execution time performance for running the five networks implemented with primitives statically allocated in the first four bars (the same primitive used across the whole network) and the optimized primitive selection with ILP on the ARM processor architectures Cortex-A15 with the right-most bar. The *direct* method is a simple optimized nested loops that implements a convolutional layer. *Im2row* and *im2col* expand the input tensor and use highly-optimized matrix multiplication libraries to perform all the convolution operations in one matrix multiplication operation, as previously described in Section 3. *Winograd* performs fast convolution by converting to the Fourier domain. Finally, our *ILP* solver finds the fastest combination of primitives across layers from the list of 57 primitive implementations (Figure 5a). These results show that the *im2row* method is very fast, but the combination of methods selected by our ILP method is significantly faster. Since there is no memory constrain in this experiment, the performance of our ILP is the same as that of PBQP [1], which we confirm.

Unlike PBQP, our ILP solver can also optimise primitive selection to achieve the best memory footprint (Figure 5b), which is important on devices with tight memory space. Optimising in the memory footprint direction, our ILP solver finds a combination with the smallest memory footprint. This combination is similar in memory requirements to the *direct* method, which theoretically has the smallest memory footprint, although much faster in inference time, as we will



(a) Performance optimization across the whole network.

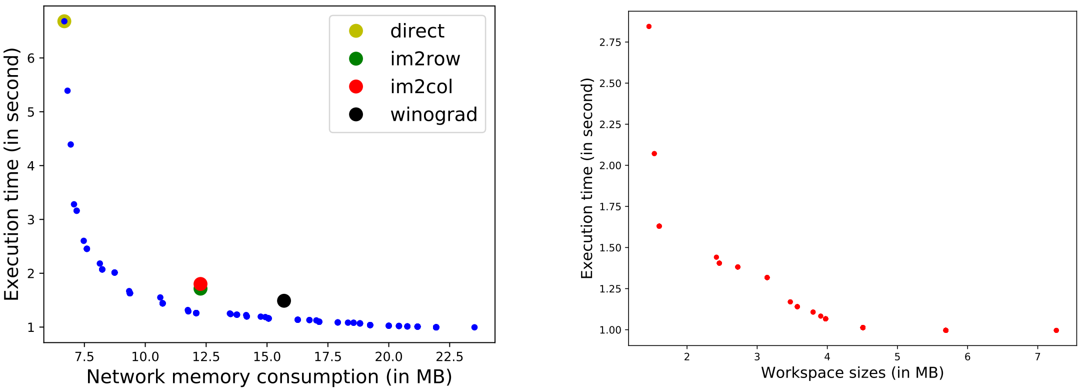
(b) Workspace-based optimization with layer level performance.

Fig. 6. Run-time performance of the two optimization strategies applied to GoogleNet (ImageNet) on the ARM A15 processor.

seen in the following section. These two experiments demonstrate the strength of our ILP solver to balance constraints and achieve the optimal points when either memory or inference time are critical for the system.

Figure 6 shows the Pareto frontier of the ILP optimal solutions for primitive selection, achieved when varying the memory budget as given condition to the solver. Figure 6a shows the inference time on a single image against the memory requirement optimised for performance across the whole network. The solutions to the left of the chart optimize for memory in detriment to execution time. Each blue dot is an optimal solution determined by IPL. We can see that the most memory efficient implementation across the whole network is achieved by the direct convolution, in detriment of inference time. Alternatively, there are many other configuration when memory constraint is relaxed. ILP determines optimal configuration points that are better in inference time than im2col, im2row and even Winograd primitives across the network, despite executing in the same memory budget. The bottom right corner is optimising purely for inference time, with almost no constraint from memory budget (the only point found by the PBQP optimizer [1]). This shows that solutions with different primitive configurations that surpass the performance of only im2col (generally regarded as the faster implementation option) do exist. Another important observation on this Pareto curve is that memory requirement can be reduced dramatically with minimal impact on inference time (more than two thirds of memory requirement can be reduced for just less than half of second inference slowdown). These observations cannot be captured by previous optimization solutions and have significant impact. Our method exposes better combinations of primitives capable of running at the same speed as im2col, but at half the memory requirement of im2col.

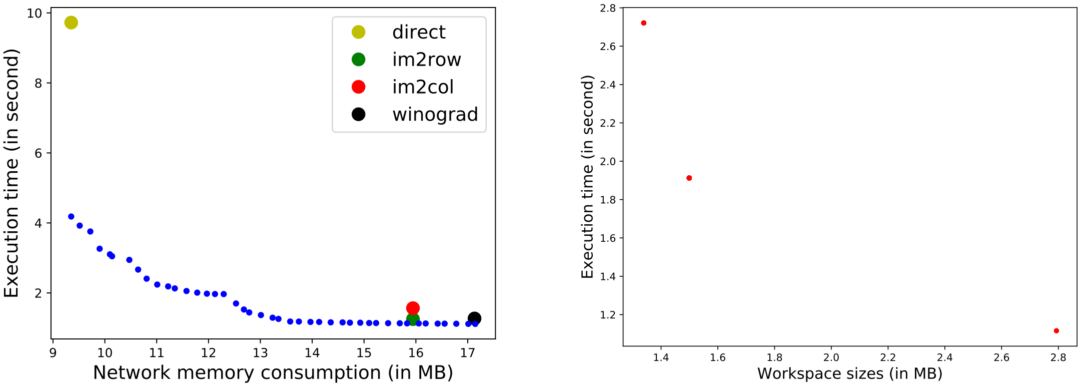
Figure 6b shows the performance of the other optimization strategy when the whole network does not fit into main memory, but with optimizations applied at layer level using the workspace memory budget strategy. This shows that primitives chosen at point A and point B (as highlighted on the figure) can execute with almost the same latency, but point B uses about 10MB less memory. Optimising for a tight memory budget (around 12MB) shows that different configurations exist that can achieve a difference of 1.4× speedup (points C and D on the figure). These observations show that even selections at layer level can have a dramatic impact on overall performance and ILP helps to expose these conditions through the Pareto trade-off curve.



(a) Across the whole network.

(b) Workspace optimization.

Fig. 7. Run-time performance of the two optimization strategies applied to AlexNet (ImageNet) on the ARM A15 processor.



(a) Across the whole network.

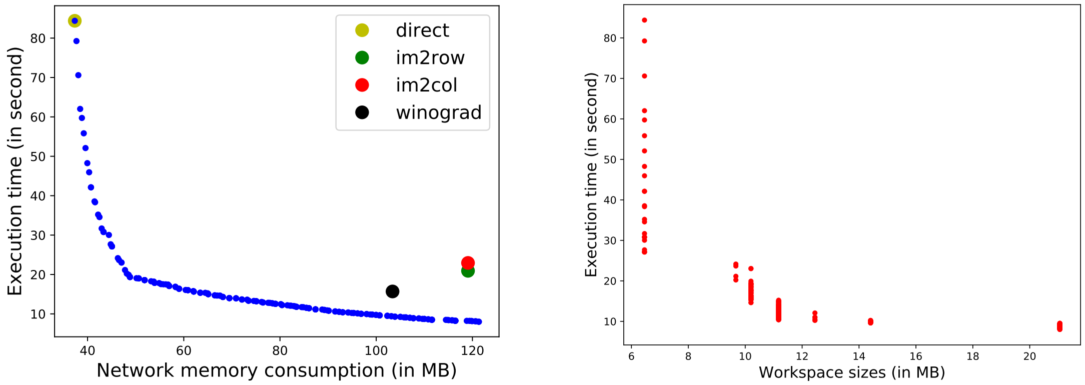
(b) Workspace optimization.

Fig. 8. Run-time performance of the two optimization strategies applied to SqueezeNet (ImageNet) on the ARM A15 processor.

We observe similar patterns in the ILP identified optimal execution points for the other deep neural networks, AlexNet (Figure 7), SqueezeNet (Figure 8), VGG (Figure 9) and ResNet, and across other processors architectures – similar patterns on the ARM Cortex A7, which we leave out in the interest of space. Our results demonstrate that our proposed ILP based primitive selection optimiser is network and hardware agnostic, identifying the best solutions under given constraints and optimisation objective (inference time, memory footprint) on any device.

5.3 Comparison with Other Methods

Here we compare our ILP-based solver with other methods for convolutional layer primitive selection. We show that our ILP method offers more control than PBQP in memory constraint conditions and finds better solutions than greedy algorithms.



(a) Across the whole network.

(b) Workspace optimization.

Fig. 9. Run-time performance of the two optimization strategies applied to VGG-D (ImageNet) on the ARM A15 processor.

5.3.1 PBQP Solver. Figure 6a shows the solutions found by ILP and PBQP [1]. All the red points are optimal solutions provided by ILP under varying execution-time and memory budget constraints, while PBQP solution is only point A, which is optimal for execution-time though completely disregarding memory footprint. Our ILP solver allows for a small sacrifice in inference time, 15%, to make a huge saving in memory footprint, 2.2× (point B). The difference between the two solvers is clear, PBQP can find only the fastest combination of primitives, but has no control over the memory footprint, which can be a crucial constraint on embedded devices.

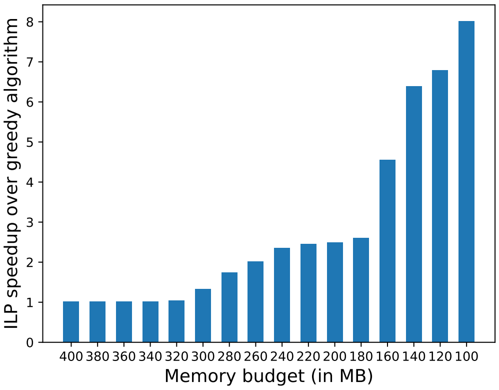


Fig. 10. Speedup from ILP versus the greedy primitive selection for GoogleNet. Moving left to right on the graphs the memory budget is **decreased**. Each bar is the speedup of the ILP optimal network arrangement for the given memory budget versus the arrangement found by the greedy heuristic selection. As the memory budget is constrained further, the higher-quality solutions found by ILP quickly outpace the solutions found by greedy heuristic selection.

5.3.2 Greedy Solver. We implemented a greedy algorithm to consider to add the memory footprint constraint to PBQP in a simple strategy. It first tries to optimize the choice of primitives to obtain the shortest inference time (same as PBQP). After the first selection of primitives per layer, if the memory footprint budget imposed to the optimizer is exceeded, the primitive with the largest memory footprint is replaced by another primitive with smaller memory footprint but fastest among the other remaining candidates, so that inference time is not compromised. This process is repeated by replacing candidate primitives until the constraint of memory budget is achieved.

In Figure 10 we compare the greedy algorithm with our ILP solver by presenting only the speedup achieved by our method against the greedy algorithm on varying memory budget constraint. This primitive selection is performed on the layers of GoogLeNet across the whole network, running on the ARM Cortex-A15 processor. We can see that while the memory budget is plentiful, both solvers make similarly good decisions (leftmost bars), while with the tightening of memory constrains (rightmost bars) the greedy algorithm begins to provide sub-optimal solutions. This is because there are just a small set of very good solutions when the memory budget is small, whereas when the memory budget increases, the greedy algorithm starts making short-sighted decisions with the simple primitive replacement strategy. In contrast, these solutions are easily identified by our ILP solver, reaching 8x speedups on the processor.

6 RELATED WORK

Several tools for producing optimized code for deep neural networks have been proposed in recent years. These tools operate at different levels of granularity, from controlling code generation for individual neurons right up to run-time heuristics for selecting full layer implementations.

At the fine-grained end of the spectrum is Latte [21], a domain-specific language, compiler and run-time for DNNs. Latte programs are written using abstractions of neurons, ensembles, and connections that allow the programmer to write a very detailed specification of the layers of a DNN.

A sophisticated compiler then produces optimized code, and pattern-matches for code regions that can be replaced by calls to optimized libraries such as GEMM. However, like most compilers, Latte can only generate code that corresponds to the program which the programmer wrote down. Our proposal, and the proposal of the prior work of Anderson et al. [1] is to allow the tool to have control of algorithmic choice for layer implementations so this burden is removed from the programmer.

Compilers for algorithmic choice have been a topic of study; one of the most widely-known general-purpose examples is the PetaBricks language and compiler from MIT [3].

More recent developments include Boda [17], a program generator for deep neural networks. Boda generates large numbers of variants of loop nests to implement DNN layers, and uses auto-tuning to select from among them. There are two significant advantages to using our approach over auto-tuning: auto-tuning takes much more time than solving a small ILP problem, and can provide at best a heuristic solution. In contrast, our analytic approach can find the global optimal network instantiation with respect to the given time and memory constraints.

While Latte and Boda can provide speedups, the majority of DNNs used in practice are still implemented using vendor-supplied programming frameworks. A common approach taken in these frameworks is to provide a means to query a heuristic cost model embedded in the library at run-time, to select the method to implement the layer which is about to be executed. This approach is taken by NVIDIA's CUDNN framework [18], among others.

In contrast, our approach is to solve for the optimal layer selection ahead of time. We profile layer execution to derive costs rather than relying on a heuristic cost model.

The TensorFlow XLA ahead-of-time compiler [6] is an optimization tool for DNNs written in the TensorFlow domain-specific language. XLA performs ahead-of-time optimizations on the network graph. The techniques we present seem well-suited to systems such as XLA, given the inherently ahead-of-time nature of our proposed approach.

7 CONCLUSION

We address a growing challenge: how to make the large deep neural network models available to use on mobile and embedded devices with limited resources. We propose an integer linear programming-based domain-specific optimizer. By selecting algorithms, implementations, and data layouts for each convolutional layer, we can optimize for both inference latency and memory footprint. Our results demonstrate how this optimizer works on five popular convolutional neural networks, observing very good performance under both constrains from inference time and memory footprint budgets when compared with greedy algorithms and performance-only solvers for convolutional layer primitive selection. We believe this opens the opportunity for many innovative edge application to build on the higher accuracy achievable only by large deep neural network models which have been unpractical on these small devices until now.

REFERENCES

[1] Andrew Anderson and David Gregg. 2018. Optimal DNN primitive selection with partitioned boolean quadratic programming. In *CGO*. 340–351.

[2] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory GEMM-based convolution algorithms for deep neural networks. *CoRR* abs/1709.03395 (2017).

[3] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. In *PLDI*.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. In *Computer Vision and Pattern Recognition*.

[5] Minsik Cho and Daniel Brand. 2017. MEC: Memory-efficient Convolution for Deep Neural Network. *CoRR* abs/1706.06873 (2017). http://arxiv.org/abs/1706.06873

[6] Jeff Dean. 2016. TensorFlow w/XLA: TensorFlow, Compiled. (December 2016). Presentation at Autodiff Workshop, co-located with NIPS 2016, Accessed: 2017-09-13.

[7] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542 (2017).

[8] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[9] S. Hodges, E. Berry, K. Wood, Ken Woodberry, and Steve Hodges. 2011. SenseCam: A wearable Camera that Stimulates and Rehabilitates Autobiographical Memory. *Memory* 19, 7 (January 2011), 685–696.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[11] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[12] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.html

[13] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. In *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

[14] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *The Conference on Empirical Methods in Natural Language Processing*.

[15] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308 http://arxiv.org/abs/1509.09308

[16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end Training of Deep Visuomotor Policies. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016).

[17] Matthew W. Moskewicz, Ali Jannesari, and Kurt Keutzer. 2017. Boda: A Holistic Approach for Implementing Neural Network Computations. In *CF* (Siena, Italy). 53–62.

[18] NVIDIA Inc 2017. *cuDNN Developer Guide*. NVIDIA Inc. Accessed: 2017-09-13.

[19] Christos H. Papadimitriou and Kenneth Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos. 2017. Fog Computing for Sustainable Smart Cities: A Survey. *ACM Comput. Surv.* 50, 3 (2017).

[21] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *PLDI* (Santa Barbara, CA, USA). 209–223.

[22] H. Wang, R. K. Katzschmann, S. Teng, B. Araki, L. GiarrÃĺ, and D. Rus. 2017. Enabling independent navigation for visually impaired people through a wearable vision-based feedback system. In *ICRA*.

[23] Ying Zhang, Mohammad Pezeshki, Philemon Brakel, Saizheng Zhang, César Laurent, Yoshua Bengio, and Aaron C. Courville. 2016. Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks. In *INTERSPEECH*.

REFERENCES

[1] Andrew Anderson and David Gregg. 2018. Optimal DNN primitive selection with partitioned boolean quadratic programming. In *CGO*. 340–351.

[2] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory GEMM-based convolution algorithms for deep neural networks. *CoRR* abs/1709.03395 (2017).

[3] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. In *PLDI*.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. In *Computer Vision and Pattern Recognition*.

[5] Minsik Cho and Daniel Brand. 2017. MEC: Memory-efficient Convolution for Deep Neural Network. *CoRR* abs/1706.06873 (2017). http://arxiv.org/abs/1706.06873

[6] Jeff Dean. 2016. TensorFlow w/XLA: TensorFlow, Compiled. (December 2016). Presentation at Autodiff Workshop, co-located with NIPS 2016, Accessed: 2017-09-13.

[7] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542 (2017).

[8] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[9] S. Hodges, E. Berry, K. Wood, Ken Woodberry, and Steve Hodges. 2011. SenseCam: A wearable Camera that Stimulates and Rehabilitates Autobiographical Memory. *Memory* 19, 7 (January 2011), 685–696.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[11] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[12] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.html

[13] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. In *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

[14] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *The Conference on Empirical Methods in Natural Language Processing*.

[15] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308 http://arxiv.org/abs/1509.09308

[16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end Training of Deep Visuomotor Policies. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016).

[17] Matthew W. Moskewicz, Ali Jannesari, and Kurt Keutzer. 2017. Boda: A Holistic Approach for Implementing Neural Network Computations. In *CF* (Siena, Italy). 53–62.

[18] NVIDIA Inc 2017. *cuDNN Developer Guide*. NVIDIA Inc. Accessed: 2017-09-13.

[19] Christos H. Papadimitriou and Kenneth Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos. 2017. Fog Computing for Sustainable Smart Cities: A Survey. *ACM Comput. Surv.* 50, 3 (2017).

[21] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *PLDI* (Santa Barbara, CA, USA). 209–223.

[22] H. Wang, R. K. Katzschmann, S. Teng, B. Araki, L. GiarrÃĺ, and D. Rus. 2017. Enabling independent navigation for visually impaired people through a wearable vision-based feedback system. In *ICRA*.

[23] Ying Zhang, Mohammad Pezeshki, Philemon Brakel, Saizheng Zhang, César Laurent, Yoshua Bengio, and Aaron C. Courville. 2016. Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks. In *INTERSPEECH*.