Mathematical Optimizations for Deep Learning

Sam Green, Craig M. Vineyard, and Çetin Kaya Koç

Abstract Deep neural networks are often computationally expensive, during both the training stage and inference stage. Training is always expensive, because back-propagation requires high-precision floating-point multiplication and addition. However, various mathematical optimizations may be employed to reduce the computational cost of inference. Optimized inference is important for reducing power consumption and latency and for increasing throughput. This chapter introduces the central approaches for optimizing deep neural network inference: pruning "unnecessary" weights, quantizing weights and inputs, sharing weights between layer units, compressing weights before transferring from main memory, distilling large high-performance models into smaller models, and decomposing convolutional filters to reduce multiply and accumulate operations. In this chapter, using a unified notation, we provide a mathematical and algorithmic description of the aforementioned deep neural network inference optimization methods.

1 Introduction

Deep neural networks (DNNs) are increasingly being incorporated into safety-critical cyber-physical systems. For example, Advanced Driver Assistance Systems use DNNs for autonomous avoidance of road hazards. Modern DNN architectures

S. Green (✉)
University of California Santa Barbara, Santa Barbara, CA, USA
e-mail: sam.green@cs.ucsb.edu

C. M. Vineyard
Sandia National Laboratories, Albuquerque, NM, USA
e-mail: cmviney@sandia.gov

Ç. K. Koç
İstinye University, İstanbul, Turkey

Nanjing University of Aeronautics and Astronautics, Nanjing, China

University of California Santa Barbara, Santa Barbara, CA, USA
e-mail: cetinkoc@ucsb.edu

Ç. K. Koç (ed.), *Cyber-Physical Systems Security*,
https://doi.org/10.1007/978-3-319-98935-8_4

require billions of floating-point multiplications and additions (MACs) for inference of a single input. Without careful design, this results in high power consumption. Fossil fuel-powered vehicles, for example, can support high energy demands, but efficient, battery-powered systems cannot. Additionally, modern large DNNs have high latency, but low latency is required for real-time cyber-physical applications. This chapter provides a unified view of the leading methods for mathematically optimized deep learning inference. The intended audience of this chapter are hardware and software researchers, as well as developers interested in efficient DNN inference. Depending on the context, "efficiency" may imply low-power or low-latency.

To motivate the need for optimizations, it is helpful to consider first-order power and silicon area requirements for DNN inference. Table 1 provides a list of energy and die area required for various operator and operand sizes. Observe that a single 32-bit floating-point multiplication (denoted "32b FP Mult") requires 20× more power and 12× more area than 8-bit integer multiplication ("8b Mult"). Also observe that the power cost of a 32-bit DRAM read is more than 100× the cost of floating-point multiplication. For this reason, efficient DNN implementations should prioritize the minimization of off-chip DRAM access first, followed by reducing operand and operator sizes. Naturally these two priorities complement one another.

DNN optimizations are useful only during the inference operation. Training a DNN requires labeled datasets and uses the back-propagation algorithm. The back-propagation algorithm uses gradient descent to make many small adjustments to the neural network weights, and these small values must be calculated and stored using full-precision accumulation. Therefore the optimizations discussed in this chapter are not primarily aimed at making training more efficient, but they are intended to make inference more efficient.

To further emphasize the need for inference efficiency, consider the number of operations required to evaluate various modern DNNs, given in Table 2. This table provides a first-order estimate for MAC and memory costs for popular

Table 1 Energy and die area costs for various operations [1]

Operation	Energy (pJ)	Area (μm)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32 FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Quantized operators and operands are preferred for low-power and low-resource applications. FP stands for floating point

Table 2 Number and cost of weights and MACs for popular deep neural network architectures

Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Weights	60k	61M	146M	138M	7M	25.5M
Read cost (8b)	10 μJ	10 mJ	23 mJ	22 mJ	1 mJ	4 mJ
Read cost (32b)	38 μJ	39 mJ	93 mJ	88 mJ	4 mJ	16 mJ
MACs	341k	724M	2.8G	15.5G	1.43G	3.9G
MAC cost (8b)	0.1 μJ	167 μJ	644 μJ	3565 μJ	329 μJ	897 μJ
MAC cost (32b)	2 μJ	3 mJ	13 mJ	71 mJ	7 mJ	18 mJ

Cost estimates are based on Table 1 and from architecture statistics provided in [1]. Note that memory costs are typically higher than MAC costs

DNN architectures. Power estimates assume 32-bit floating-point arithmetic and are derived from Table 1. MAC costs capture the power requirement for each network to perform the necessary operations for providing a single inference. The memory cost is best case and assumes weights are read from DRAM only once per inference; actual memory costs will be higher if intermediate results must be transferred back to DRAM during inference of the network. In Table 2 note that even though the number of MACs is much greater than the number of weights, the high DRAM read cost results in the power consumed between the two to be roughly equivalent.

The process of DNN training may be thought of as an exploration over a parameter space to find values which will solve an inference task. As will be expanded on, the weights found using standard training methods result in DNNs which are over-parameterized, which means they have redundancy. When the DNN performs satisfactorily during cross validation, back-propagation is no longer needed, and optimizations may be applied to decrease parameter redundancy. The goal of mathematical optimizations for deep learning is to find the most compact network which performs satisfactorily at its assigned real-world inference tasks.

DNN architectures are composed of various layer types: convolutional, fully connected, dropout, pooling, and others. Each layer type was developed to solve a particular weakness, and each classification problem is best solved by a different architecture, or combination of layers. Convolutional and fully connected layers represent the greatest computational expense in DNN inference, and optimizing these layer types is the focus of this chapter. Both convolutional and fully connected layers require repeated multiplication and addition, but they typically use different algorithmic steps. Adapting notation of [2], we represent an L-layer DNN as $\langle \mathrm{I}, \mathrm{W}, \mathrm{O} \rangle$, where:

- $\mathrm{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$ and $\mathrm{W}_l \in \mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ are layer l's input tensors and weight tensors, respectively. c_{in} represents the number of **input channels** and c_{out} represents the number of **output channels**.[1] x and y are the width and height of each input channel, and w and h are the width and height of each filter.

[1] Also called input filter maps (ifmaps) and output filter maps (ofmaps) in literature.

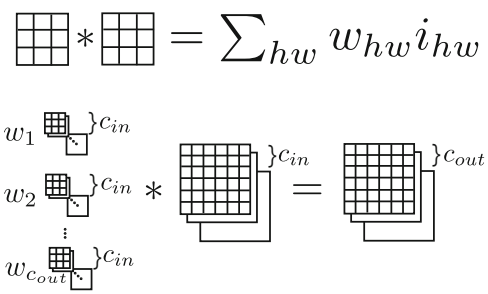

Fig. 1 Convolutional layers convolve a weight filter with an input. Filters are usually 5×5, 3×3, or 1×1. Each step of the convolution involves multiplying and accumulating elements of the weight filter with a **receptive field** of the input. The top illustration represents the basic convolution operation (∗). The lower illustration represents c_{out}, c_{in}-channel filters which are convolved with a c_{in}-channel input tensor, which results in an c_{out}-channel output tensor

- $O_l \in \{*, \cdot, \text{other}\}$ specifies whether the layer's operation type is convolution (∗), fully connected (·), or some other less computationally expensive type.

Convolutional layers convolve a $\mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ weight filter tensor with a $\mathbb{R}^{c_{in} \times x \times y}$ input tensor, where (w,x) and (h,y) represent the widths and heights of the two respective tensors and may be different sizes and c_{in} and c_{out} represent the number of input and output channels. In particular the (w, h) for weight filters are often smaller than the (x, y) for inputs. c is the number of channels in the given layer; this value is equal for both the weight filter tensor and input tensor. As illustrated in Fig. 1 (top), each step in the convolution requires a sum of products between elements of the weight filter and elements of the receptive field of the input filter.

Note that what is shown in Fig. 1 (top) only depicts convolution of a single channel. If there are multiple channels, then the summation is also over all channels. Figure 1 (bottom) shows a higher-level view, where each c_{in}-channel weight filter is convolved with the c_{in}-channel input tensor. When multiple channels are included in the convolution, each output of the convolution becomes the triple sum across the channels. The number of weight filters in a layer equals the number of channels in the output tensor: if there are c_{out} weight filters, there will be c_{out} channels in the output tensor.

Computation for fully connected layers requires a single matrix-vector product. The input tensor $I_l \in \mathbb{R}^{c_{in} \times x \times y}$ is flattened to a vector $\in \mathbb{R}^{c_{in} \cdot x \cdot y}$. The weight tensor is denoted $W \in \mathbb{R}^{w \times h}$, where $w = c_{in} \cdot x \cdot y$ (from the input tensor dimensions) and h is equal to the number of desired output units from the fully connected layer. An illustration of a fully connected layer is given in Fig. 2.

After a weight filter W is convolved with an input I in a convolutional layer, or the matrix-vector product between weights and layer inputs is produced for a fully connected layer, the resulting matrix of vector entries is typically passed through a nonlinearity function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$. A commonly used nonlinearity is the rectified

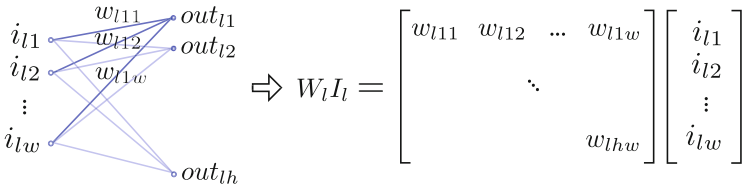

Fig. 2 Fully connected layers flatten the input tensor into a vector and multiply by a weight matrix with the same number of columns as the vector and as many rows as desired

linear unit (ReLU), which is defined as:

$$\sigma_{\text{ReLU}}(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{else.} \end{cases} \tag{1}$$

But more extreme nonlinearities exist, such as the binarized activation function which outputs only two values, −1 and 1:

$$\sigma_b(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{else.} \end{cases} \tag{2}$$

The choice of nonlinearity function influences the performance and computational cost of inference. Specifically, using the binarized activation function can lead to the elimination of floating-point and fixed-point arithmetic during inference, as detailed in Sect. 3.2.

Both convolutional and fully connected layers require many memory access and MAC operations, but a variety of numerical optimizations may be applied to DNN inference. Some optimizations reduce power and some optimizations reduce both power and latency. Furthermore, it is possible to optimize a DNN and maintain classification accuracy, but there also exist extreme optimization methods which result in unavoidable accuracy loss. Depending on the application, decreased accuracy may be worth the reduction in power and latency.

The remainder of this chapter provides an introduction to the common approaches of DNN mathematical optimization. The approaches are grouped by five primary strategies:

- **Pruning**: reduces the number of weights, which, in turn, reduces the total number of MAC operations, amount of traffic required to transfer weights, and storage requirements. This method applies to fully connected and convolutional layers.
- **Quantization**: lowers the number of bits of precision representing neural network inputs, weights, or activations, which lowers both memory requirements and silicon required for processing elements. This method applies to fully connected and convolutional layers.

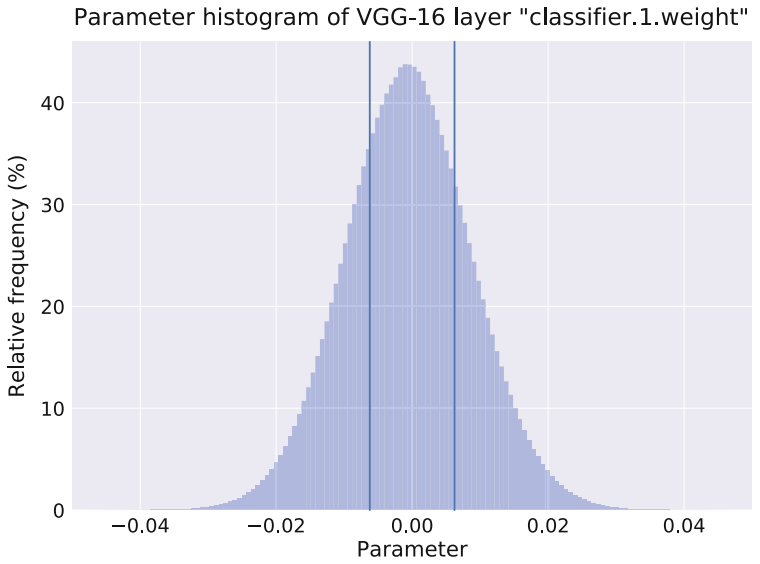

Fig. 3 Histogram of weights of the first fully connected layer in VGG-16. The name "classifier.1.weight" corresponds to the VGG-16 implementation found in torchvision [4]. The two vertical lines correspond to thresholds of values smaller than the 50th-percentile. These values may be pruned (permanently set to zero) and the remaining values fine-tuned with no loss in accuracy [3]. The same procedure may be applied to all other layers in the network

- **Weight Sharing and Compression**: forces weights to share values, thus decreasing memory storage and traffic. This method applies to fully connected and convolutional layers.
- **Model Distillation**: the training of a smaller network to mimic the behavior of larger network, reducing the number of weights and lowering latency. This method applies to fully connected and convolutional layers.
- **Filter Decomposition**: modifies convolutional filter designs such that the number of weights and latency is reduced. This method only applies to convolutional layers.

2 Pruning

Pruning applies to fully connected and convolutional layers and eliminates each layer's smallest weights, which has the consequence of reducing the number of MAC operations, the amount of traffic required to transfer weights, and storage requirements. The typical procedure is to train the network until the desired accuracy is reached and then to prune the smallest pth-percentile of weights by setting them to zero. Pruning is followed by fine-tuning the remaining weights, which can be accomplished using the same dataset as used during initial training.

In [3], the authors report 9× and 13× reduction in weights for AlexNet and VGG-16 with no impact on test accuracy. A histogram of the normalized frequency of weights is given in Fig. 3, where the smallest 50th-percentile is delineated with two vertical lines. In practice, one would pick the percentile threshold for each layer heuristically, that is, the percentile threshold would be a hyperparameter for each layer. This process is represented in Algorithm 1.

Algorithm 1 Pruning

Require: L-layer DNN $\langle \mathrm{I}, \mathrm{W}, \mathrm{O}, \mathrm{P} \rangle$, where I_l and W_l are layer l's input tensors and parameter tensors respectively, and O_l specifies whether the layer's type is convolutional, fully-connected (or some other type), and P is the pruning percentile for each layer.

Ensure: Pruned and fine-tuned network weights W.

1. Initial training:

Perform standard training of DNN until satisfactory performance is achieved.

2. Pruning:

For each layer l in $\langle \mathrm{I}, \mathrm{W}, \mathrm{O}, \mathrm{P} \rangle$, eliminate weights in W_l which are less than layer l's p^{th} percentile, where $p = \mathrm{P}_l$.

3. Fine-tuning:

Perform standard (re)training of remaining weights W, until maximum performance is achieved.

After pruning, the resulting DNN will be sparse, with many weights set to zero. Standard architectures, like GPUs, are currently not designed to take advantage of sparsity and will perform multiplication regardless if one of the operands is zero. In order to benefit from pruning, the architecture must be designed in such a way as to take advantage of sparsity. This will add edge cases to standard logic design. For example, consider a product summation tree, which can parallelize MAC operations. Even if the tree is designed to ignore products with a zero operand, it must still take into account that the zero product must be passed to the next tree level at the appropriate time. Recently, architectures for handling sparse dataflows have been developed. One such architecture reduces the amount of "wasted" logic required for ignoring zero products by only passing non-zero products to processing elements downstream [5].

3 Quantization

Before 2015, most DNNs were trained using 32-bit floating-point arithmetic. In this section we summarize approaches for using reduced precision, or quantized arithmetic, for DNN inference. Quantization reduces the amount of weight data that must be transferred from DRAM to processing elements. Additionally, quantized arithmetic is less expensive in terms of power and silicon area than full-precision arithmetic. Quantization may be applied to weights, activations, or both weights and activations. We emphasize that quantization techniques using <16 bits

currently only provide efficiency benefits during inference, because back-propagation requires accumulation of small values, and therefore $\geq$16 bits.

It appears that 8-bit or 16-bit quantization is adequate for most DNN inference tasks. For example, Google's DNN accelerator, the Tensor Processing Unit (TPU), exclusively uses 8-bit or 16-bit integer arithmetic [6]. The TPU (and the successor TPUv2) is becoming a critical component of Google's computing ecosystem. Additionally, NVIDIA's Pascal architecture was designed to support 16-bit floating-point and 8-bit integer arithmetic.

In this section we focus on extreme quantization methods which binarize weights and activations. Binarization usually has a large negative impact on performance, but we present techniques in Sects. 3.1 and 3.2 which reduce the impact.

Note that in this section, we will sometimes use a unified notation which applies to both convolutional and fully connected layers. In a convolutional layer, a c-channel weight filter $\mathrm{W} \in \mathbb{R}^{c \times w \times h}$ is convolved with an input $\mathrm{I} \in \mathbb{R}^{c \times w \times h}$. Convolution is performed by $\mathrm{W} * \mathrm{I}$. At a specific receptive field, the core operation may be interpreted as the inner products between vectors. In this section, we sometimes use the notation $\mathrm{W}^\top \mathrm{I}$ to denote the convolution of a filter with a specific receptive field. Simultaneously, the $\mathrm{W}^\top \mathrm{I}$ notation captures the partial calculation of a fully connected layer.

3.1 Binary Weights

In 2015, BinaryConnect [7] was an early DNN quantization method and exemplifies the field's approach to quantization. During inference, BinaryConnect quantizes full-precision DNN weights W to $\{-1, 1\}$, using the sign function:

$$w^{(b)} = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{else.} \end{cases} \tag{3}$$

Equation (3) discards real-valued information, but, in doing so, it also eliminates the need for floating-point multiplication during inference. Instead, signed floating-point addition may be used for unit activation input calculations. During back-propagation, the error caused by quantization is used to update the real-valued Ws. After training is complete, full-precision weights and arithmetic are no longer required and may thereafter be discarded. From a hardware perspective, memory overhead is 32$\times$ less when using BinaryConnect-derived weights. However, this technique has an accuracy cost. When using the AlexNet DNN architecture, BinaryConnect achieves 61% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using AlexNet with 32-bit full-precision accuracy [2].

In Algorithm 2 we outline the steps of BinaryConnect. Note here that we separate the bias terms from W, where normally it is included in that tensor for notation convenience. The reason here is that the bias is always added, even with

full-precision arithmetic, so there is no benefit to quantize it. Also note the `clip` function in Algorithm 2 limits the full-precision weights to between [−1, 1].

Algorithm 2 BinaryConnect [7]

Require: Inputs I, targets y, previous full-precision weights W, biases b, learning rate η, and objective function J.
Ensure: Updated $\{-1, 1\}$-valued weights $\mathrm{W}^{(b)}$ and real-valued bias b.

1. Forward propagation:
$\mathrm{A}_0 = \mathrm{I}$
for $l = 1$ to L
  for k^{th} filter in l^{th} layer
    $\mathrm{W}_{lk}^{(b)} \leftarrow \texttt{binarize}(\mathrm{W}_{lk})$ using Eq. (3)
    $\mathrm{A}_{lk} \leftarrow \mathrm{W}_{l}^{(b)} * \mathrm{A}_{(l-1)k} + b_{lk}$
2. Backward propagation:
Initialize output layer's activation gradient $\frac{\partial J}{\partial A_L}$ using y, A_L, and J
for $l = L$ to 2
  for k^{th} filter in l^{th} layer
    Compute $\frac{\partial J}{\partial A_{(l-1)k}}$ knowing $\frac{\partial J}{\partial A_{lk}}$ and $\mathrm{W}_{lk}^{(b)}$
3. Update weights:
Compute $\frac{\partial J}{\partial \mathrm{W}_{lk}^{(b)}}$ and $\frac{\partial J}{\partial b_{lk}}$, knowing $\frac{\partial J}{\partial A_{lk}}$ and $A_{(l-1)k}$
$\mathrm{W} \leftarrow \texttt{clip}(\mathrm{W} - \eta \frac{\partial J}{\partial \mathrm{W}})$
$b \leftarrow b - \eta \frac{\partial J}{\partial b}$

Not made explicit in Algorithm 2 is how the gradient signal passes through the binarization function given in Eq. (3). This is required for calculation of $\partial J / \partial \mathrm{W}_{lk}^{(b)}$. We cannot merely take the derivative of the binarization function, because it is 0 everywhere except at $\mathrm{W} = 0$, where the function is discontinuous. To handle this, the authors used a variant of the **Straight-Through Estimator** (STE) during back-propagation [8]. The modified STE is defined as:

$$\texttt{STE}(\text{pre-binarized value}) = \begin{cases} 0 & \text{if } x < -1, \\ 1 & \text{if } -1 \geq \text{pre-binarized value} \leq 1, \\ 0 & \text{if } x > 1. \end{cases} \tag{4}$$

During back-propagation, instead of flowing through the binarization function, the incoming gradient signal is multiplied by the value of `STE`, which is evaluated at the pre-binarized weight value (or pre-activation value, when using XNOR-Net, discussed below). Clipping caused by multiplying by the `STE` has the effect of canceling the gradient when the pre-binarized value is too large.

To summarize BinaryConnect, we take the sign of the real-valued weights during inference. During back-propagation, the errors caused by binarization may be very small (with significant changes accumulating over many inputs), and we track those small changes in full-precision versions of the weights. After training is complete, the full-precision weights may be discarded, only keeping their sign information.

XNOR-Net [2] introduced a method which is almost identical to BinaryConnect, but it performs binarization in way which achieves higher accuracy. As with BinaryConnect, weights are binarized during inference, but then they are also scaled by a factor which attempts to compensate for the binarization. Specifically, XNOR-Net introduced the following approximation for the inner product[2]:

$$\mathrm{W}^\top \mathrm{I} \approx \alpha \mathrm{W}^{(b)\top} \mathrm{I}, \tag{5}$$

where $\mathrm{W}^{(b)}$ is the binarized version of W using Eq. (3). This notation is slightly different than that used in Algorithm 2, where we are able to binarize the entire W tensor at once. But with XNOR-Net, each filter in each convolutional layer requires a separate α. To keep the notation simple, separate filters are not denoted.

To find the optimal scaling factor α, we solve the following optimization problem:

$$\begin{aligned} J(\alpha) &= \left\| \mathrm{W} - \alpha \mathrm{W}^{(b)} \right\|^2, \\ \alpha^* &= \arg\min_{\alpha} J(\alpha). \end{aligned} \tag{6}$$

That is, we are seeking an α which minimizes the distance between W and $\alpha \mathrm{W}^{(b)}$. For intuition, consider a scalar w and its binarized version $w^{(b)}$; in this case $\alpha = w/w^{(b)}$ perfectly minimizes the distance between w and $w^{(b)}$. Expanding the norm in Eq. (6) gives:

$$J(\alpha) = \alpha^2 \mathrm{W}^{(b)\top} \mathrm{W}^{(b)} - 2\alpha \mathrm{W}^\top \mathrm{W}^{(b)} + \mathrm{W}^\top \mathrm{W}. \tag{7}$$

We now take the derivative of $J(\alpha)$ with respect to α, set it to zero, and solve for α:

$$\frac{dJ(\alpha)}{d\alpha} = 2\alpha \mathrm{W}^{(b)\top} \mathrm{W}^{(b)} - 2\mathrm{W}^\top \mathrm{W}^{(b)}. \tag{8}$$

Let $n = \mathrm{W}^{(b)\top} \mathrm{W}^{(b)}$, which is also equal to the number of weights in the binarized filter. Substituting n into Eq. (8) and solving for α gives α^*:

$$\alpha^* = \frac{\mathrm{W}^{(b)\top} \mathrm{W}^{(b)}}{n} = \frac{\mathrm{W}^{(b)\top} \mathrm{sign}(\mathrm{W})}{n} = \frac{\sum |\mathrm{W}|}{n}. \tag{9}$$

New α^*s must be calculated every time W changes, i.e., each time back-propagation is used to update the weights, but, after the training is completed, α^* may be saved for use during inference.

[2]Note that we consider W and I to be flattened.

Table 3 The XNOR operation captures the behavior of signed multiplication

Signed multiplication		
Inputs		Output
I_i	W_i	$I_i \times W_i$
−1	−1	1
−1	1	−1
1	−1	−1
1	1	1

XNOR "multiplication"		
Inputs		Output
I_i	W_i	$\overline{I_i \oplus W_i}$
0	0	1
0	1	0
1	0	0
1	1	1

Using the weight binarization methods above, we may eliminate most multiplications from inference,[3] and instead we only need signed addition. If we assume 32-bit multiplication and addition, this results in 32× power reduction for weight transfer from DRAM and ∼3× power reduction for arithmetic. When using the AlexNet DNN architecture, XNOR-Net (binary weights, full-precision activations) achieves 79.4% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using AlexNet with 32-bit full-precision accuracy [2]. We next consider operator optimizations which become available when both weights *and* inputs are binarized.

3.2 Binary Weights and Activations

If weights and activations are binarized, then we are able to eliminate almost all floating-point (and fixed-point) calculations, resulting in extreme energy savings. Specifically, when weights and inputs are binarized, the XNOR operation[4] may be used to calculate inner products during inference [9]. The XNOR logic truth table is given on the right in Table 3. The left-hand side provides the truth table for signed multiplication between scalar values $I_i \in I$ and $W_i \in W$. Note that by mapping −1 to 0, the two tables give identical output.

XNOR logic is simple and efficient to implement in hardware and may be used as the multiplication operator for the calculation of inner products during inference. To use the XNOR "product" between I and W for the input into a unit's nonlinearity function, we first map all −1s to 0s and then calculate the XNOR values for both vectors. The Hamming weight[5] (HW) of the XNOR vector result is then compared to $\#bits/2$, where $\#bits$ is the size of W and I. If the Hamming weight is greater than or equal to $\#bits/2$, then output 1, otherwise output 0. Note that after the initial mapping of −1 to 0, we no longer need to map back to −1 during the remainder of the inference procedure.

BinaryNet [9] operates similarly to BinaryConnect, with the addition that activations are also binarized. When using BinaryNet, the activation inputs are summed, as

[3] Multiplication by α is still necessary when using the weight binarization technique in XNOR-Net.

[4] Not to be confused with XNOR-Net [2]. Here we are referring to the exclusive-NOR operation.

[5] Hamming weight is defined as the number of 1s in a vector.

with BinaryConnect, and then the resulting sum is converted to [−1, 1] using the sign function. This optimization eliminates all full-precision calculations and replaces them with signed integer calculations. As with BinaryConnect, BinaryNet requires full-precision gradient updates during training, and during back-propagation the `STE` function (Eq. (4)) is used for both the activation and weights. BinaryNet achieves 50.42% top-5 accuracy on AlexNet, compared to 80.2% accuracy when using the same DNN topology and 32-bit full-precision accuracy [2].

XNOR-Net also has a version which binarizes both weights and activations. Similar to XNOR-Net's weight-only binarization presented above, there is a scaling factor α which may (optionally) be used to reduce the error between full-precision and binarized dot products:

$$\begin{aligned} J(\alpha) &= \left\| \mathrm{I}^{\top}\mathrm{W} - \alpha \mathrm{I}^{(b)\top}\mathrm{W}^{(b)} \right\|^2, \\ \alpha^* &= \arg\min_{\alpha} J(\alpha). \end{aligned} \tag{10}$$

This is solved in the same manner as Eq. (6), giving:

$$\alpha^* = \frac{\sum |\mathrm{I}^{(b)\top}\mathrm{W}^{(b)}|}{n} = \frac{\sum |\mathrm{I}||\mathrm{W}|}{n}. \tag{11}$$

Note that a separate scaling factor α^* must be solved for each receptive field and weight filter combination *both during training and when using the neural network after training*. This high computational overhead limits the use of vanilla XNOR-Net. Fortunately, in practice, the authors of BinaryNet found that the scaling factor for binarized weights was much more important than the scaling factor for binarized inputs and may therefore be ignored. We summarize the weight-scaled version of XNOR-Net with the following algorithm:

Similar to the calculation of $\partial J/\partial \mathrm{W}_{lk}^{(b)}$ in Algorithm 2, both partial derivatives $\partial J/\partial \mathrm{W}_{lk}^{(b)}$ and $\partial J/\partial \mathrm{A}_{lk}^{(b)}$ in Algorithm 3 are multiplied by the `STE` function in Eq. (4), where the inputs to `STE` are the real-valued weight and activation, respectively.

XNOR-Net using binarized inputs and weights achieves 69.2% accuracy on AlexNet, compared to BinaryNet's 50.42%, and full-precision accuracy of 80.2%. The XNOR-Net and BinaryNet papers introduce other training tips for improved performance. The aggregate contributions of the performance techniques introduced in XNOR-Net likely account for its significant gain over BinaryNet.

4 Weight Sharing and Compression

Top-performing neural networks use millions of weights which are typically transferred from DRAM to processing elements for inference (see Table 2). When these weights are transferred, DRAM energy cost can surpass arithmetic cost for

Algorithm 3 (Weight-scaled) XNOR-Net [9]

Require: Inputs I, targets y, previous full-precision weights W, biases $\boldsymbol{b}$, learning rate η, and objective function J.

Ensure: Updated $\{-1, 1\}$-valued weights $\mathrm{W}^{(b)}$, weight scaling factors $\boldsymbol{\alpha}$, and real-valued bias $\boldsymbol{b}$.

1. Forward propagation:

$\mathrm{A}_0 = \texttt{binarize}(\mathrm{I}_0)$

for $l = 1$ to L

  for k^{th} filter in l^{th} layer

    $\alpha_{lk} = \frac{1}{n}||\mathrm{W}_{lk}||_{\ell 1}$

    $\mathrm{W}_{lk}^{(b)} \leftarrow \texttt{binarize}(\mathrm{W}_{lk})$ using Eq. (3)

    $\mathrm{A}_{lk}^{(b)} \leftarrow \texttt{binarize}\big((\alpha_{lk}\mathrm{W}_{lk}^{(b)}) * \mathrm{A}_{(l-1)k}^{(b)} + b_{lk}\big)$ using Eq. (3)

2. Backward propagation:

Initialize output layer's activation gradient $\frac{\partial J}{\partial A_L}$ using y, A_L, and J

for $l = L$ to 2

  for k^{th} filter in l^{th} layer

    Compute $\frac{\partial J}{\partial A_{(l-1)k}^{(b)}}$ knowing $\frac{\partial J}{\partial A_{lk}^{(b)}}$ and W_{lk}

3. Update weights:

Compute $\frac{\partial J}{\partial \mathrm{W}_{lk}^{(b)}}$ and $\frac{\partial J}{\partial b_{lk}}$, knowing $\frac{\partial J}{\partial A_{lk}^{(b)}}$ and $A_{(l-1)k}$

$\mathrm{W} \leftarrow \texttt{clip}(\mathrm{W} - \eta \frac{\partial J}{\partial \mathrm{W}^{(b)}})$

$b \leftarrow b - \eta \frac{\partial J}{\partial b}$

performing a single inference. Weight sharing clusters weights into shared values and is applied after the network has reached peak performance. Once weights have been clustered, compression may be used to transmit cluster indices instead of full-precision values. Weight sharing coupled with compression is a method to retain the high performance typically provided by large full-precision neural networks while simultaneously reducing the amount of data sent over DRAM [3].

4.1 Weight Sharing

To apply weight sharing, first, the DNN is trained to maximum performance using standard training methods. After training, each layer's weights are grouped into clusters, where the number of weights in a layer is much greater than the number of clusters. After assigning weights to clusters, the network goes through a retraining phase.

For example, consider Fig. 4 which illustrates a 4×4 channel from some weight filter in W. Assume that the filter is part of a trained network. To apply weight sharing, we use k-means clustering [3], which assigns the weights $w \in \mathrm{W}$ to m cluster assignments $C^\star = \{c_1, c_2, \ldots, c_m\}$, such that the within-cluster sum of squares is minimized:

$$C^\star = \arg\min_{C} \sum_{i=1}^{m} \sum_{w \in c_i} |w - c_i|^2. \tag{12}$$

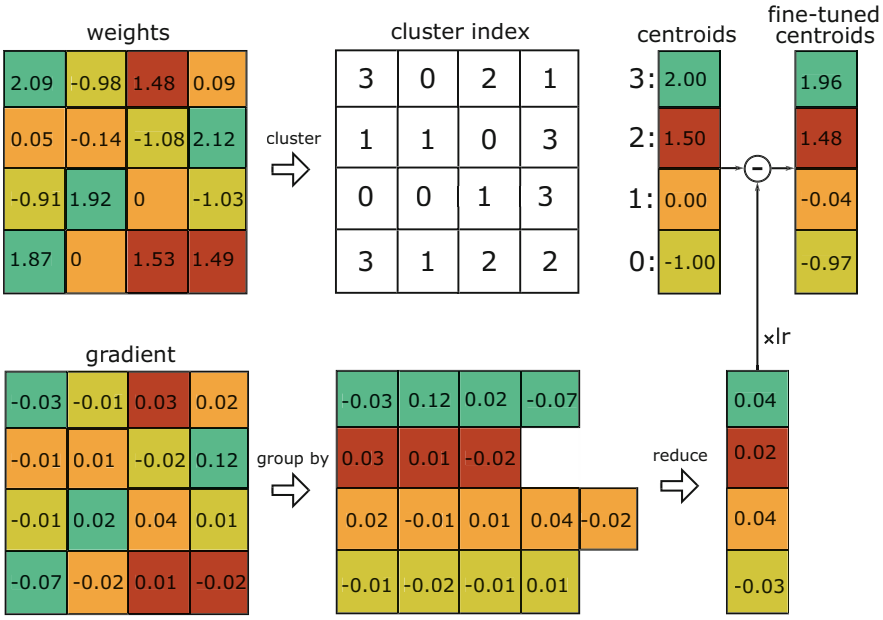

Fig. 4 After training, 16 weights have been clustered into 4 centroids. From that point on, clustered weights are equal to their centroid. Partial derivatives are calculated with respect to the weight values, as usual, but the gradients are accumulated and subtracted from the centroids [3]

After assignment to clusters, we calculate the centroids $\tilde{w}_i$ of each cluster c_i by taking the average value of each cluster:

$$\tilde{w}_i = \frac{1}{|c_i|} \sum_{w \in c_i} w. \tag{13}$$

In Fig. 4, $m = 4$, and the top portion of the plot illustrates 16 weights and their associated clusters and centroids.

After clustering, weights in the original filter are replaced by their centroid value (this is represented by the shading in Fig. 4). Next, the clustered weights are fine-tuned by reusing the original training data. The key difference between standard training and the fine-tuning phase is how the weights are updated during gradient descent (GD). In GD each weight is moved a small amount in the direction which will improve an objective function, e.g.:

$$\mathrm{W}_{l,w} = \mathrm{W}_{l,w} - \eta \frac{\partial J(\mathrm{W})}{\partial \mathrm{W}_{l,w}}. \tag{14}$$

However, after clustering, we apply GD to the centroid value of each weight cluster. For example, suppose the centroid $\tilde{w}_i$ of weight cluster c_i is to be updated using

GD. To update centroid $\tilde{w}_i$, we use the sum of partial derivatives with respect to weights assigned to that cluster:

$$\tilde{w}_i = \tilde{w}_i - \eta \sum_{w \in c_i} \frac{\partial J(\mathrm{W})}{\partial w}. \tag{15}$$

The lower portion and the subtraction in Fig. 4 illustrate the gradient descent step of back-propagation when using clustering. After the fine-tuned centroids have been calculated, they will replace the previous weight values in each cluster. The update given in Eq. (15) is repeated until maximum performance is attained.

Algorithm 4 Weight sharing

Require: Inputs I, previously *trained* full-precision weights W, number of clusters m, learning rate η, objective function J.
Ensure: Clustered and fine-tuned weights W
1. Cluster assignment:
for $l = 1$ to L
for k^{th} filter in l^{th} layer
Assign weights in filter k to m clusters using Eq. (12):
$C^\star \leftarrow \texttt{knn}(\mathrm{W}_{lk}, m)$
Replace weights in each cluster with centroid value using Eq. (13):
$\mathrm{W}_{lk} \leftarrow \texttt{centroid}(\mathrm{W}_{lk}, C^\star)$
2. Inference:
Perform standard inference using centroid-mapped weights.
3. Fine-tuning:
Calculate standard partial derivatives with respect to weights $\frac{\partial J(\mathrm{W})}{\partial w}$.
Update centroid values by summing partial derivatives in each cluster and using gradient decent: $\tilde{w}_i = \tilde{w}_i - \eta \sum_{w \in c_i} \frac{\partial J(\mathrm{W})}{\partial w}$
Replace weights in each cluster with updated centroid values.
4. Optionally repeat:
Repeat steps 2 and 3 until objective function is optimized.

The steps for weight sharing are provided in Algorithm 4. The algorithm is written from the perspective of CNNs, but adapting it for other DNN designs only requires clustering the appropriate values. For example, the values in fully connected layer could be clustered.

After weight values have been clustered and fine-tuned, there is an opportunity to decrease the storage and traffic requirements for loading the DNN weights from memory to an accelerator. This process is detailed in the following subsection.

4.2 Compression

Weight sharing reduces the amount of data transmitted over DRAM by intentionally creating redundancy in the form of a cluster index. For example, in Fig. 4 we see

that 16 original values are represented by four cluster values. Redundancy created by weight sharing is exploitable with compression methods [3].

If a network uses b bits of precision, then a full-precision network with n weights requires nb bits of transmission. After weight sharing, only a single full-precision value (the centroid) must be transmitted for each cluster; this results in mb bits. The indices for m clusters are represented with $\log_2(m)$ bits; therefore transmitting n indices requires $n\log_2(m)$ bits. In general, n weights clustered into m clusters compress the weights by a factor of:

$$\frac{nb}{n\log_2(m) + mb}. \tag{16}$$

For example, referring to Fig. 4, and assuming 32-bit floating-point weights, we see that $nb = 16 \cdot 32$ and $n\log_2(m) + mb = 16 \cdot 2 + 4 \cdot 32$. Therefore, by using weight sharing and compression, we reduce the traffic by a factor of 352.

5 Model Distillation

Large neural networks have a tendency to generalize better than smaller networks. Similarly, ensemble methods combine the predictions of multiple algorithms, e.g., DNNs, random forests, SVMs, logistic regression, etc., and almost always outperform the predictions from an individual algorithm. Both large networks and ensemble methods are attractive from an accuracy perspective, but many applications cannot support the time or energy it takes to perform inference using such approaches. Model distillation is the training of a smaller, more efficient, DNN to predict with the performance close to a larger DNN or ensemble [10, 11].

When training a multiclass network, first, the softmax of network logits a_i is used to calculate class probabilities:

$$\hat{y}_i = \frac{e^{a_i/T}}{\sum_{j=1}^{|C|} e^{a_j/T}}, \tag{17}$$

where C is the set of classes which the network can identify and T is the temperature and is usually set to 1. Class probabilities are then used in the cross-entropy error function:

$$J(y, \hat{y}) = -\sum_{i=1}^{|C|} y_i \log \hat{y}_i, \tag{18}$$

where y is the correct training label for a given input and $\hat{y}$ is the vector of class prediction probability output from the network. Using standard supervised training, y is a one-hot encoded vector, with 1 in the position of the correct label, and 0

everywhere else. Therefore, when the correct class is $i = k$, Eq. (18) simplifies to:

$$J(\hat{y}, k) = -\log \hat{y}_k. \tag{19}$$

Equation (19) contains the objective function typically differentiated during the training of a large neural network.

The output probabilities of a previously trained large network capture rich information not available in the original training set, which only contain input examples and the correct label for each input. For example, assume a classification dataset which includes cars, trucks, and other non-vehicle classes. During training, when learning instances of car classes, only a single correct label (y, which is one-hot encoded) will be used. Once trained, if presented with a previously unseen photo of a car, the car and truck class probabilities will most likely both contain significant information regarding the correct class, whereas the potato class probability would not contain as much information. Model distillation uses all of this information.

There are various techniques to implement distillation. Initially, assume a large network has been trained to high performance, and a smaller network is to be trained with distillation. Additionally, assume we do not have access to the correct training labels. In this case, we may input random images into the large network and use *all* of its prediction probabilities $\hat{y}$ as a **soft target** for the distilled network's output $\tilde{y}$:

$$J(\tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i. \tag{20}$$

This is similar to Eq. (18), except $y = \hat{y}$, and we have class probabilities for each entry in $\hat{y}$, so it does not simplify to Eq. (19).

If training labels are also available, the objective function can be improved by summing Eqs. (18) and (20), giving:

$$J(y, \tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \alpha \hat{y}_i \log \tilde{y}_i + \beta y_i \log \tilde{y}_i, \tag{21}$$

where α is a hyperparameter which sets the relative importance for matching soft targets and β sets the relative performance for selecting the correct class. In practice [11] found that α should be higher than β.

In addition to hyperparameters α and β, [11] also found that the temperature in Eq. (17) impacts distillation performance. Higher temperatures make "softer" probability distributions. To understand why this may be important, consider the logits $[1, 2, 10]$, which have a softmax with $T = 1$ of $[1 \times 10^{-4}, 3 \times 10^{-4}, 9.995 \times 10^{-1}]$. The small probabilities slow down learning during back-propagation. However, when $T = 10$, the softmax becomes $[0.22, 0.24, 0.54]$, which has ranges that will cause learning to occur more quickly with back-propagation. It can therefore be useful to use high T values for the softmax of both the large network and distilled

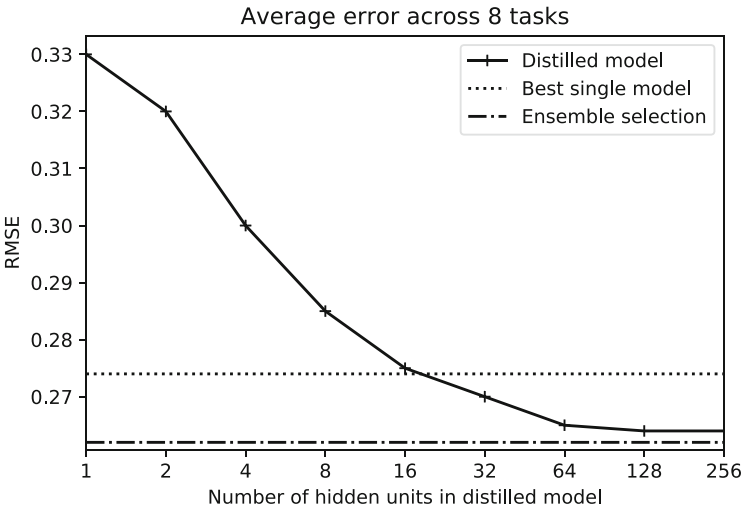

Fig. 5 An ensemble of models was trained for eight classification tasks. Distillation was then used to train a neural network to behave like each ensemble. The plot to the right compares average performance between the ensemble of classifiers, the best individual classifier in each ensemble, and the distilled classifiers. Once the distilled classifier has enough capacity, its average approaches the ensemble average [10]

network during the distillation phase.[6] After distillation is finished, T may be reset to 1.

Distillation is effective for transferring information from trained large networks to untrained smaller networks. In [11], a large DNN was trained to classify MNIST, resulting in 67 test errors. A smaller network, trained and tested with the same sets as the larger network, resulted in 146 errors. However, when the smaller network was trained with distillation, it only made 74 test errors.

Thus far we have discussed how to distill a DNN into a smaller network. Similar methods may be used to distill an ensemble of classifiers. In [10], eight binary classification problems were solved by an ensemble of methods, and then a neural network was trained by distillation to capture the behavior of the ensemble. The average performance of the small distilled model is given in Fig. 5. It can be seen that the average performance of the distilled model was similar to a giant ensemble prediction derived from SVMs, bagged trees, boosted trees, boosted stumps, simple decision trees, random forests, neural nets, logistic regression, k-nearest neighbor, and naive Bayes.

A smaller distilled model is obviously guaranteed to be more efficient than a large DNN or ensemble of models, and the distillation approaches presented in

[6]The softmax layer is at the output and has no trainable weights. It can therefore be replaced in the larger network with a separate temperature, with no need for retraining.

this section are a promising avenue to achieving adequate performance, given hard resource constraints. The steps for distillation are summarized in Algorithm 5.

Algorithm 5 Distillation

Require: Inputs I, optional targets y, previously *trained* high performance network $\langle \mathrm{W}, \mathrm{O} \rangle_{large}$, *untrained* distilled network $\langle \mathrm{W}, \mathrm{O} \rangle_{dist}$

Ensure: Trained distilled network $\langle \mathrm{W}, \mathrm{O} \rangle_{dist}$

1. Inference:

$\hat{y} \leftarrow$ output probabilities of $\langle \mathrm{I}, \mathrm{W}, \mathrm{O} \rangle_{large}$

$\tilde{y} \leftarrow$ output probabilities of $\langle \mathrm{I}, \mathrm{W}, \mathrm{O} \rangle_{dist}$

2. Calculate loss:

if targets y are available

$J(y, \tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i + y_i \log \tilde{y}_i$

else

$J(\tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i$

3. Update distilled model weights:

$\mathrm{W}_{dist} \leftarrow \mathrm{W}_{dist} - \eta \nabla_{\mathrm{W}_{dist}} J$

4. Optionally repeat:

Repeat steps 2 and 3 until objective function is optimized.

6 Filter Decomposition

AlexNet introduced the first popular high-performance convolutional neural network (CNN) architecture, which has since been widely adopted and modified [12]. The AlexNet architecture won fame by winning the 2012 ImageNet Challenge, which required classification across 1000 categories. AlexNet uses five convolutional layers, three fully connected layers, and other less computationally expensive layers. Modern CNNs use even more convolutional layers, for example, Google's GoogLeNet-v1 CNN architecture uses 57 convolutional layers, but only one fully connected layer.

Fully connected layers are expensive from a bandwidth perspective, because they perform only one multiply-accumulate operation (MAC) per byte transferred over memory. Convolutional layers, however, are efficient from a bandwidth perspective, but they are expensive computationally. For example, AlexNet's three fully connected layers require 58.6M MAC operations and 58.6M weights, whereas AlexNet's six convolutional layers require 666M MAC operations and only 2.3M weights. The total cost of a fully connected layer or convolutional layer is the total number of MACs plus the total number of bytes required for the layer.[7] The choice of filter sizes in convolutional layers has a large impact on the bandwidth and computational costs of a CNN. In this section we analyze the bandwidth and computational impacts of different convolutional filter designs.

[7] First-order estimates of power costs can be calculated using Table 1.

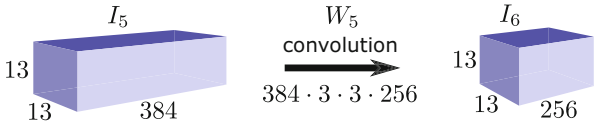

Fig. 6 Example calculation of `MAC cost` of the fifth convolution in AlexNet. For intuition in understanding `MAC cost`, consider that each point in I_6 is the result of applying a $384 \times 3 \times 3$ filter tensor to I_5. Therefore the total number of MACs needed to calculate I_6 is $256 \times 13 \times 13 \times 384 \times 3 \times 3$. This is a different perspective on the calculation than given in the main text

We loosely base our discussion on AlexNet, because it is well understood and the foundation of modern CNN designs. AlexNet convolutional layers use three filter shapes, 11×11, 5×5, or 3×3, and four channel depths, 96, 256, or 384. The shape of convolution filters has a significant impact on computational cost. To calculate the MAC cost for layer l's convolution operations, we first recall the notation introduced in Sect. 1, where layer l's filter tensor is denoted $\mathrm{W}_l \in \mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ and layer l's input tensor is denoted $\mathrm{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$. The number of MAC operations in a convolutional layer is found by[8]:

$$\texttt{MAC cost} = \texttt{cardinality}(\mathrm{I}_l) \times \frac{\texttt{cardinality}(\mathrm{W}_l)}{c_{in} \text{ from } \texttt{cardinality}(\mathrm{W}_l)}, \tag{22}$$

where `cardinality()` returns the number of elements in the input tensor. The bandwidth required for a filter, assuming 32-bit floating-point weights, is calculated as:

$$\texttt{Byte cost} = c_{in} \times w \times h \times c_{out} \times 4 \text{ bytes}. \tag{23}$$

The goal of efficient CNN design is to obtain the highest classification performance, using the fewest number of MACs and weights. Therefore from an efficiency perspective, the cost of CNN inference is:

$$\texttt{COST}() = c_1 \texttt{MAC cost} + c_2 \texttt{Byte cost} + c_3 \texttt{CNN errors}, \tag{24}$$

where the coefficients c depend on the priorities and budget of the CNN's designer.

To better understand Eq. (22), consider the calculation of the number of MACs in the fifth convolutional layer of AlexNet, illustrated in Fig. 6. In this case $\texttt{cardinality}(\mathrm{I}_5) = 384 \times 13 \times 13$ and $\texttt{cardinality}(\mathrm{W}_5) = 384 \times 3 \times 3 \times 256$. So the total number of MAC operations for $\mathrm{I}_5 * \mathrm{W}_5$ is $384 \times 13 \times 13 \times 3 \times 3 \times 256 = 150\text{M}$. Additionally, the size of W_5 is $384 \times 3 \times 3 \times 256 = 885\text{k}$ weights.

[8] Our calculations assume there is no pooling layer after convolution, which is now commonly the case.

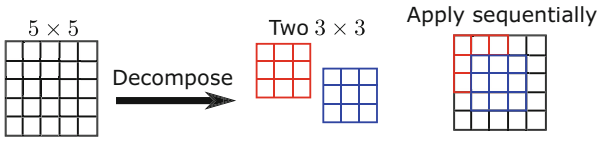

Fig. 7 A "large" convolutional filter may be separated into two smaller filters, which retain the feature detection capabilities of the larger filter. The outputs of the smaller filters are summed. This approach is used to reduce the number of bytes required to represent filters and to reduce the number of MAC operations

As another example, assume that instead of 3×3 filters, 5×5 filters were used in AlexNet's fifth convolutional layer. 5×5 filters cause the number of MAC operations to increase to 415M and byte cost to increase to 2.5 MB. A larger filter can capture more detail, and suppose that switching to a 5×5 filter increased classification accuracy, but caused the total cost to exceed the time and energy budget allotted to the CNN. Perhaps surprisingly, there are techniques to extract the benefit of 5×5 filters without using 5×5 filters.

The concept of filter decomposition was introduced in [13], where two smaller filters $\mathrm{W}_{l,1}$ and $\mathrm{W}_{l,2}$ were applied to the input tensor I_l and then added (prior to the nonlinearity), giving $\mathrm{I}_{l+1} = \mathrm{I}_l * \mathrm{W}_{l,1} + \mathrm{I}_l * \mathrm{W}_{l,2}$. As shown in Fig. 7, instead of using a single 5×5 filter tensor in the previous case, two 3×3 tensors can be used. Specifically, instead of performing $256 \times 13 \times 13 \times 5 \times 5 \times 384 = 415\text{M}$ MAC operations (requiring 2.5M weights), $256 \times 13 \times 13 \times 3 \times 3 \times 384 \times 2 = 300\text{M}$ MACs are performed (requiring 1.8M weights). Similarly, a 5×1 and 1×5 filter may be used, requiring $256 \times 13 \times 13 \times 5 \times 2 \times 384 = 166\text{M}$ MAC operations (requiring 1M weights), which is close to the original 150M MACs and 885k weights required when using a single 3×3 filter tensor.

Going even further, [14] introduced 1×1 convolutions, which are used to create **bottleneck layers**, because they can shrink an input tensor. 1×1 filters detect correlation between corresponding weights in each channel, which may be seen when considering their full notation: $c_{in} \times 1 \times 1 \times c_{out}$. For example, suppose we are given input $\mathrm{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$, then a filter $\mathrm{W}_l \in \mathbb{R}^{c_{in} \times 1 \times 1 \times c_{out}}$ may be chosen such that $c_{out} \ll c_{in}$. Convolving W_l with I_l gives $\mathrm{I}_{l+1} \in \mathbb{R}^{c_{out} \times x \times y}$. The information from I_l is not lost, even though I_{l+1} now has fewer channels than I_l. 1×1 convolutions capture channel correlations, compared to larger filters which capture channel and spatial correlations.

Various filter schemes can be combined. For example, a 1×1 convolution may be followed by a 3×3 or 5×5 convolution. The goal here is to extract channel correlations using the 1×1 convolution and to extract spatial (and channel) correlations using the 3×3 or 5×5 filter. Going back to our original AlexNet example, we calculated the number of MACs used for the convolution of I_5 and W_5 as $256 \times 13 \times 13 \times 3 \times 3 \times 384 = 150\text{M}$ MACs and 885k weights. We can reduce this by picking a smaller c_{out} size for W'_5, e.g., 64, giving $256 \times 13 \times 13 \times 1 \times 1 \times 64 = 2.8\text{M}$ MACs and 16k weights. We may then add another convolution layer, using a $384 \times 3 \times 3$ filter W'_6, and return to the original shape of I_6 using

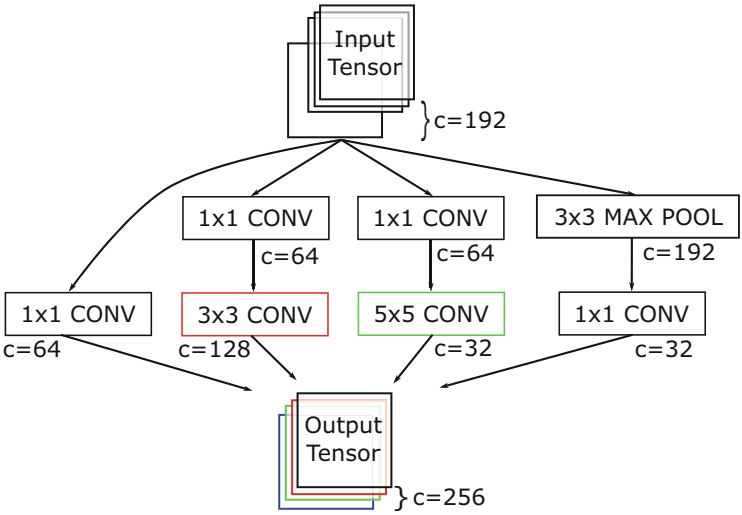

Fig. 8 Diagram of an Inception module. Layer inputs are passed through separate 1×1 bottleneck layers and then through standard convolutional layers. This technique allows for the use of different filter sizes, without paying the computational or bandwidth cost of normal convolutional layer implementations [15]

$384 \times 13 \times 13 \times 3 \times 3 \times 64 = 37$M MACs and 221k weights. We now have extracted both channel and spatial correlations, using 1×1 and 3×3 filters and a total of 39.8M MACs and 237k weights, much fewer than the original example which used 150M MACs and 885k weights. Bottleneck layers followed by convolution have proven to be an effective way to increase efficiency without sacrificing accuracy.

Filter decomposition represents a fundamentally different way to improve DNN inference efficiency, compared to earlier sections. Specifically, by making careful architectural choices, high performance can be maintained and fewer weights and MAC operations can be used. The methods introduced here may also be combined. For example, Inception is a modern CNN architecture, which combines bottleneck layers and various filter shapes to capture the benefits of every possible combination. Figure 8 illustrates an Inception module, which combines many convolutional layers and outputs each combination as stacks of sub-channels. Without the 1×1 bottleneck layers, such an architecture would be much more expensive.

7 Conclusion

Deep neural networks are increasingly being integrated into cyber-physical systems, which have power, silicon area, latency, and accuracy budgets. This chapter introduced various mathematical and algorithmic methods for optimized DNN inference:

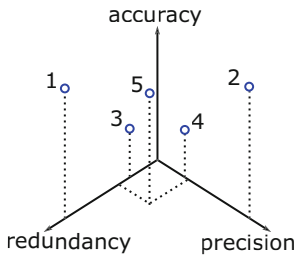

Fig. 9 The notional trade-off between accuracy, redundancy, and precision. In general, one may prioritize any two at the expense of the third [16]. There is currently no formal proof for this plot, but most of the optimization papers referenced in this chapter report metrics across the different axes and seem to generally follow the trend of this plot

- Eliminating "small" weights via pruning, which reduces the required number of multiply-accumulate operations
- Quantization, or reducing the precision, of layer inputs and/or weights to reduce computation and data transfer costs
- Sharing weights between layer units and therefore enabling data transmission compression
- Training small models to mimic larger models by distilling the information from the larger models into the smaller models
- Separating larger convolutional filters into smaller filters while retaining the performance of the larger filters

These optimization methods may be used individually or may be combined for greater optimization. Note that the methods are not equivalent and should be expected to affect performance metrics in different ways.

Unfortunately most of the optimizations introduced here will result in an accuracy loss when compared to a high-performance model which was designed with no regard to computational efficiency. The trade-off between accuracy, redundancy, and precision is depicted in Fig. 9 [16]. In general, one may expect to obtain high accuracy when using high-precision (e.g., floating-point) arithmetic (Pt. 2 in Fig. 9) and lower accuracy when using low-precision arithmetic (Pt. 4). But low-precision arithmetic may be offset with redundancy (e.g., larger models) (Pt. 1). Likewise the errors caused by using low-redundancy (few weights) models may be offset, to some extent, with high-precision arithmetic.

Ultimately, it is the DNN architect's task to find a design which achieves minimum acceptable performance, given a particular resource (e.g., latency, silicon area, power) budget. The methods introduced in this chapter facilitate this task.

Acknowledgements Sandia National Laboratories is a multimission laboratory managed and operated by the National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the US Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

1. D. William, High-performance hardware for machine learning, in *Conference on Neural Information Processing Systems* (2015)
2. M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, Xnor-net: Imagenet classification using binary convolutional neural networks, in *European Conference on Computer Vision* (2016)
3. S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding, in *International Conference on Learning Representations* (2016)
4. S. Marcel, Y. Rodriguez, Torchvision the machine-vision package of torch, in *International Conference on Multimedia* (ACM, New York, 2010), pp. 1485–1488
5. A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, W.J. Dally, Scnn: an accelerator for compressed-sparse convolutional neural networks, in *International Symposium on Computer Architecture* (ACM, New York, 2017), pp. 27–40
6. N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T.V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C.R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D.H. Yoon, In-datacenter performance analysis of a tensor processing unit, in *International Symposium on Computer Architecture* (ACM, New York, 2017), pp. 1–12
7. M. Courbariaux, Y. Bengio, J.-P. David, Binaryconnect: training deep neural networks with binary weights during propagations, in *Conference on Neural Information Processing Systems* (2015)
8. G. Hinton, Neural networks for machine learning. https://www.coursera.org/learn/neural-networks (2012). Accessed 03/14/18
9. I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, in *Conference on Neural Information Processing Systems* (2016), pp. 4107–4115
10. C. Buciluǎ, R. Caruana, A. Niculescu-Mizil, Model compression, in *International Conference on Knowledge Discovery and Data Mining* (ACM, New York, 2006), pp. 535–541
11. G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network (2015). Preprint. arXiv:1503.02531
12. A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems* (2012), pp. 1097–1105
13. R. Rigamonti, A. Sironi, V. Lepetit, P. Fua, Learning separable filters, *Conference on Computer Vision and Pattern Recognition* (IEEE, New York, 2013), pp. 2754–2761
14. M. Lin, Q. Chen, S. Yan, Network in network, *International Conference on Learning Representations* (2014)
15. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in *Conference on Computer Vision and Pattern Recognition* (IEEE, New York, 2015)
16. D. Strukov, ECE594BB neuromorphic engineering, University of California, Santa Barbara, March (2018)