

Accelerating Sparse Deep Neural Networks

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic,
Dusan Stosic, Ganesh Venkatesh, Chong Yu, Paulius Micikevicius

NVIDIA

{asitm, jalbericiola, jpool, darkos, dstosic, chongy, paulium}
@nvidia.com

Abstract. As neural network model sizes have dramatically increased, so has the interest in various techniques to reduce their parameter counts and accelerate their execution. An active area of research in this field is sparsity – encouraging zero values in parameters that can then be discarded from storage or computations. While most research focuses on high levels of sparsity, there are challenges in universally maintaining model accuracy as well as achieving significant speedups over modern matrix-math hardware. To make sparsity adoption practical, the NVIDIA Ampere GPU architecture introduces sparsity support in its matrix-math units, Tensor Cores. We present the design and behavior of Sparse Tensor Cores, which exploit a 2:4 (50%) sparsity pattern that leads to twice the math throughput of dense matrix units. We also describe a simple workflow for training networks that both satisfy 2:4 sparsity pattern requirements *and* maintain accuracy, verifying it on a wide range of common tasks and model architectures. This workflow makes it easy to prepare accurate models for efficient deployment on Sparse Tensor Cores.

1 Introduction

In the area of Deep Learning, using larger neural network models typically leads to higher accuracy for various tasks [1–4]. Modern state-of-the-art models can consist of hundreds of billions of parameters and require trillions of compute operations per input sample. Pruning of neural network parameters has emerged as an important technique to reduce model sizes and compute requirements at inference time.

Pruning accomplishes this by pushing certain parameter values to zero, inducing sparsity in a model [5–13]. However, existing pruning methods can struggle to simultaneously maintain model accuracy and gain inference performance (speed). Fine-grained sparsity maintains accuracy but poorly utilizes memory accesses and fails to take advantage of modern vector and matrix math pipelines, thus it does not outperform traditional dense models on processor architectures such as GPUs. Coarse-grained sparsity can better utilize processor resources but fails to maintain accuracy beyond moderate sparsity ratios.

In this paper, we describe a 2:4 (read as “two-to-four”) sparsity pattern that halves a model’s parameter count, requiring that every group of consecutive four values contains at least two zeros. We also describe a workflow to prune traditional, dense models for this pattern while maintaining their accuracy. This workflow prunes weights of a densely-trained model once, then repeats the training session with a fixed sparsity pattern using the same hyper-parameters as in the original training session. Furthermore, we describe Sparse Tensor Cores, introduced in the NVIDIA Ampere GPU architecture [14], to accelerate operations on 2:4 sparse matrices. Sparse Tensor

Cores double math throughput for matrix-multiply operations when the first argument is a compressed 2:4 sparse matrix. Matrix multiplication is a compute primitive behind math-intensive neural network operations such as convolutions, linear layers, recurrent cells, and transformer blocks.

The contributions of this paper include: (1) a fine-grained 2:4 structured pruning approach, (2) a compression format to efficiently store such pruned tensors in memory, (3) hardware architecture to accelerate sparse matrix multiplication involving a 2:4 sparse tensor, and (4) an empirically-verified workflow that re-trains pruned weights to eliminate accuracy loss in many standard networks. We summarize prior sparsity research in Section 2. The 2:4 pattern, compressed storage format required by Sparse Tensor Cores, and Sparse Tensor Core operation are detailed in Section 3. Section 4 describes the methodology for training neural networks to have 2:4 weight sparsity so that inference can be accelerated. In Section 5, we provide an empirical study on a variety of popular tasks and neural network architectures highlighting the universality of our proposed workflow in maintaining model accuracy while not having to change any hyper-parameters. Section 6 concludes the paper with a summary and directions for future work.

2 Related Work

Neural network model pruning approaches can be grouped into the following categories:

- train and prune a dense model, then fine-tune the remaining weights in the model to recover accuracy,
- train a dense model with gradual pruning to obtain a sparse model,
- train a sparse model with a sparsity pattern selected *a priori*, or
- train a sparse model with a sparsity pattern determined based on trained dense version.

Fine-tuning methods prune fully-trained dense weights and continue to fine-tune the remaining weights for additional training samples. Different approaches in this space can be distinguished by the pruning method, pruning schedule, sparsity structure, and fine-tuning schedule used during the fine-tuning phase. Pruning methods typically eliminate weights using weight magnitude based metrics [7] or some salience-based criteria [15]). A variety of pruning schedules have been proposed – single step [16] and gradual/iterative [10, 11, 17]. Iterative methods prune weights gradually over a number of steps, in each step eliminating either a fixed number of weights [6, 7] or choosing a fraction of weights based on an analytical function [9]. Furthermore, the pattern used to prune models may adhere to a specific structure [10, 17–19] or follow no structure at all [7, 11]. Structured pruning removes parameters in groups (entire filters, channels, etc.) in order to exploit hardware and software optimized for dense computation. However, at higher levels of sparsity these pruning methods lose model accuracy [10, 20–23]. For example, ResNet-50 can see a 2× speedup through channel pruning, but close to 1.5% accuracy is lost [10]. Unstructured pruning eliminates individual parameters without any regard to the resulting pattern. Networks pruned with unstructured sparsity tend to retain more accuracy than similarly sized networks pruned with structured sparsity, but they rarely fully utilize the underlying hardware capabilities. For example, model parameters can be pruned by nearly 13× with no loss in accuracy, but the pruning pattern is not conducive to hardware acceleration [7]. Hence, the performance benefit with such unstructured pruning approaches is negligible and at times negative, even when pruning rate is high (e.g. 95%) [23]. The difficulty of getting inference speed benefits with unstructured patterns is even more pronounced on processor architectures with matrix pipelines, such as GPUs and TPUs (Tensor Cores and systolic arrays, respectively). No universal fine-tuning schedule has been proposed yet – schedules often vary from model to model (for example, using 10 epochs of fine-tuning for one network but requiring 20 epochs for another network model [10]).

Some approaches train models with a fixed sparsity pattern starting from the random initialization of the network, but the pattern is computed based on a fully trained [13] or partially trained [24, 25] dense model of the same architecture. These approaches maintain accuracy and achieve 80%-99% sparsity. While run-time is not discussed in these works, given the use of unstructured sparsity, it is unlikely for these pruned models to efficiently utilize modern matrix processors.

Sparse models can also be obtained by starting from randomly-initialized dense models and gradually introducing sparsity as training progresses [9, 21, 26–31]. Typically, 70%-99% unstructured sparsity is targeted to prepare models for inference. As described above, these models struggle to outperform dense models due to their unstructured sparsity. Furthermore, to maintain accuracy, model and training hyper-parameters are often modified, and these modifications can be quite model-specific. For example, in [26], the hidden size of a recurrent network was increased by $1.75\times$ while pruning to 95% sparsity. In another case [31], the training schedule was extended for some networks by $5\times$ in order to reach the same accuracy as the dense model.

Finally, there have been proposals to design sparse model architectures, aimed at efficient hardware utilization during training and inference. Most prominent example of this category is block sparsity, where sparsity is introduced at a block level allowing good utilization of matrix math pipelines [32]. However, similar to approaches discussed above, in order to maintain accuracy, one has to increase the hidden size compared to the dense model. Block sparsity has found use for cases where using a larger hidden size enables higher accuracy but is impractical with dense models. Additionally, there has been work investigating fine-grained structured sparsity and motivating the need to prune in a fine-grained pattern that is conducive to hardware acceleration [17, 33]. Key points are optimized GPU kernels to speed up such pruned models on CUDA cores [17] and the benefits of custom hardware to speed up structurally pruned models [33].

Summary: While a variety of approaches for pruning neural networks to high degrees of sparsity have been proposed, no one method has been described to maintain accuracy while achieving inference speedup. Fine-tuning workflows provide inconclusive results on which pruning schemes to use (e.g. magnitude or heuristic based weight pruning, prune partially trained or fully trained weights), what pruning method to follow (e.g. one-shot or iterative pruning, layer by layer pruning or whole network pruning.) and what fine-tuning schedule to use (e.g. how many epochs of fine-tuning, what learning rate to use, etc.). In short, extracting performance from hardware by pruning networks while maintaining accuracy and using a fine-tuning workflow that is consistent across a variety of networks is still an open problem. This motivated our search for a sparsity pattern that enables hardware acceleration as well as maintains accuracy with a workflow applicable over a wide range of neural network tasks and models.

3 Sparsity Support in the NVIDIA Ampere Architecture

We introduce 2:4 sparsity to address the challenges of adopting sparsity outlined in the last section. The 2:4 pattern mandates that for each group of 4 values, at least 2 must be zero. This leads to 50% sparsity, which makes maintaining accuracy without hyper-parameter exploration much more practical than, say, 80% sparsity. When applied to a matrix, the 2:4 pattern has the following benefits over alternative sparsity approaches:

- efficient memory accesses
- a low-overhead compressed format
- 2x math throughput increase on the NVIDIA Ampere GPU architecture

The pattern and compressed format are detailed in Section 3.1, while Sparse Tensor Cores that take advantage of this format are described in Section 3.2.

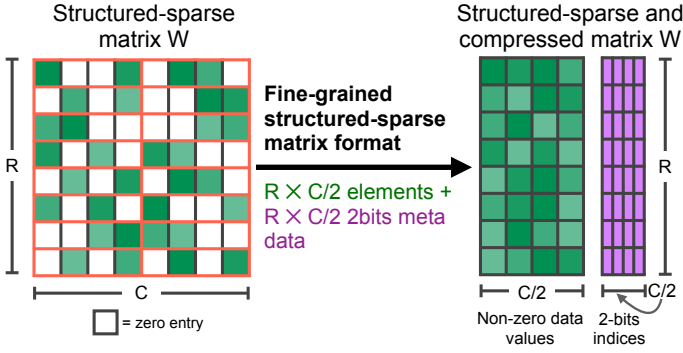


Fig. 1. Structured-sparse matrix (W) storage format. The uncompressed matrix is of dimension $R \times C$ and the compressed matrix is of dimension $R \times \frac{C}{2}$.

3.1 2:4 Sparsity and Its Benefits

An example of a matrix that satisfies 2:4 sparsity pattern requirement is shown in Figure 1. With this pattern, only the 2 nonzero values in each group of 4 values need to be stored. Metadata to decode compressed format is stored separately, using 2-bits to encode the position of each nonzero value within the group of 4 values. For example, metadata for the first row of matrix in Figure 1 is $[[0, 3], [1, 2]]$. Metadata information is needed to fetch corresponding values from the second matrix when performing matrix multiplication. Note that for a group of 4 values having more than 2 zeros, the compressed format will still store 2 values to maintain a consistent format.

Efficient memory accesses: Unstructured sparsity patterns lead to poor utilization of cache lines when accessing memory, thus under utilizing memory bandwidth. Furthermore, unstructured patterns commonly use CSR/CSC/COO storage formats [34], which lead to data-dependent accesses, thereby increasing latency for matrix reads. In contrast, 2:4 sparsity has the same level of sparsity at every sub-block of the larger matrix, which enables hardware to fully-utilize large memory reads. Similarly, since the sparsity is constant across the matrix, there is no indirection required; a nonzero value’s position in memory can be determined from the compression rate directly.

Compressed format efficiency: Using CSR format for unstructured sparsity can introduce storage overhead due to metadata of up to 200% (consider 8b quantized weight values for inference: column-index for the value would require 16-bits or more for even modestly-sized matrices). Due to its 4-value block size, the 2:4 sparse storage format (shown in Figure 1) requires only 2-bits metadata per value, limiting storage overhead to 12.5% and 25% for 16b and 8b values, respectively. For 16-bit operands, storing a sparse tensor in compressed format leads to $\sim 44\%$ savings in storage capacity: 4 dense elements require $4 \times 16 = 64$ -bits of storage while 2:4 sparsity leads to 2×16 -bits + 2×2 -bits = 36-bits to store the two non-zero elements. For 8-bit operands, storing in compressed format saves $\sim 38\%$ in memory capacity and bandwidth compared to the dense tensor.

3.2 Structured-Sparse GEMM on Tensor Cores

Tensor Cores, first introduced in the NVIDIA Volta GPU architecture, accelerate matrix-multiply-and-accumulate (MMA) instructions that are fundamental to neural network layers involving math operations such as convolutions, linear layers, recurrent cells, and transformer blocks.

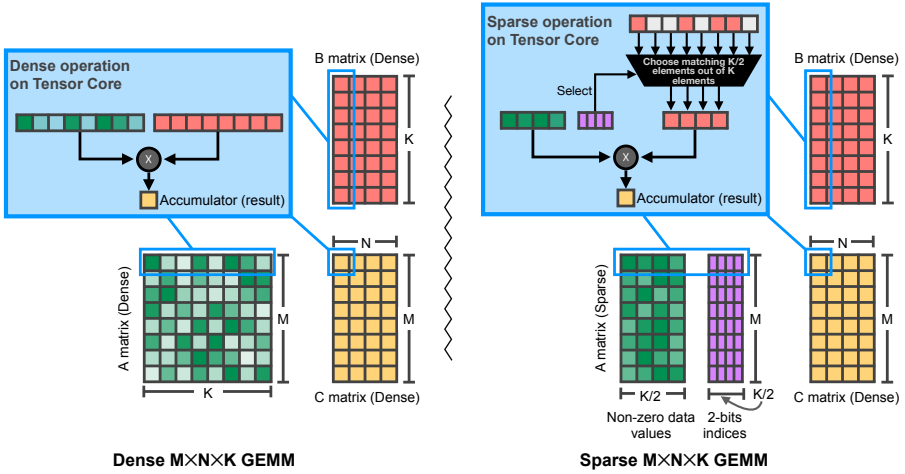


Fig. 2. Mapping a $M \times N \times K$ GEMM onto a Tensor Core. Dense matrix A , of size $M \times K$, (left side) becomes $M \times \frac{K}{2}$ (right side) after pruning with 2:4 sparsity. Sparse Tensor Core hardware selects only the elements from B that correspond to the nonzero values in A , skipping the unnecessary multiplications by zero. In both dense and sparse GEMMs, B and C are dense $K \times N$ and $M \times N$ matrices, respectively.

Table 1. A100 Tensor Core input/output formats (numeric precision) and performance in tera-operations per second (TOPS).

Input Operands	Accumulator	Dense TOPS	Sparse TOPS
FP32	FP32	19.5	-
TF32	FP32	156	312
FP16	FP32	312	624
BF16	FP32	312	624
FP16	FP16	312	624
INT8	INT32	624	1248

The NVIDIA Ampere GPU architecture extends the Tensor Cores to also handle 2:4 sparsity by allowing the first argument be stored in the sparse format described in Section 3.1. Thus, Sparse Tensor Cores perform sparse matrix \times dense matrix = dense matrix operation (the second input matrix and the output matrix are dense). Figure 2 shows how a 2:4 sparse GEMM operation is mapped to Tensor Cores. 50% sparsity on one of the operands halves the required multiply-and-add operations, resulting in (up to) a $2\times$ performance increase over equivalent dense GEMMs. Sparse Tensor Cores support FP16, BF16, and 8b-integer input/output types. Furthermore, TF32 mode is supported for FP32 input/output but the pattern becomes 1:2 sparse. Peak dense and sparse Tensor Core throughputs are shown in Table 1.

It is the application’s responsibility to ensure that the first operand is a matrix stored in the compressed 2:4 format. cuSPARSElt and other libraries provide APIs for compression and sparse math operations, while, starting in version 8.0, the TensorRT SDK performs these functions for 2:4 sparse weights automatically. NVIDIA libraries require that input dimensions of a sparse matrix multiplication be multiples of 16 and 32 for 16-bit (FP16/BF16) and 8b-integer formats, respectively.

Speedups that 2:4 sparse matrix multiplications achieve over dense multiplications depend on several factors, such as arithmetic intensity and GEMM dimensions. Fig-

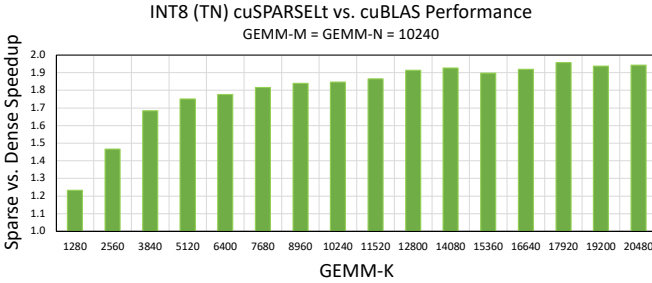


Fig. 3. Comparison of sparse and dense INT8 GEMMs on NVIDIA A100 Tensor Cores. Larger GEMMs achieve nearly a $2\times$ speedup with Sparse Tensor Cores.

ure 3 shows speedups achieved over a sampling of GEMM dimensions (cuSPARSElt¹ cuBLAS² libraries were used for the sparse and dense GEMMs, respectively). As larger GEMMs tend to have higher arithmetic intensity, they get closer to the $2\times$ speedup afforded by Sparse Tensor Cores. For language modeling networks, N is often the sequence length times the batch size: for a sequence length of 256, one would need a batch size of 40 to see this plot with N equal to 10K. M and K are related to the hidden dimensions of the layers in the network, which is typically scaled up to increase network accuracy; GPT-3 [35], for example, uses a hidden size of 12,288.

4 Network Pruning Workflow

In this section, we describe a workflow that prunes a network with the 2:4 sparsity pattern, maintains original accuracy, and avoids any hyper-parameter searches. Since our aim is to reduce neural network size and run-time at deployment, we trade a higher training cost for a simple and general workflow – the additional training cost can be amortized over the deployment lifetime of days to months.

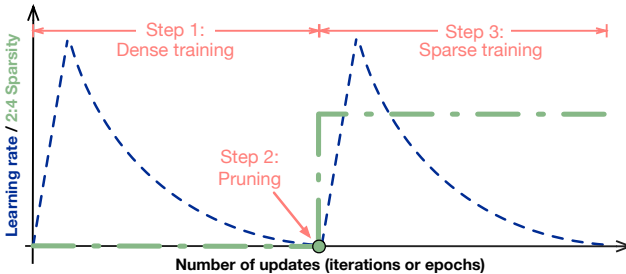


Fig. 4. A typical network training process adapted for 2:4 sparsity: dense training (step 1) is followed by a pruning stage (step 2) and a second, identical training stage (step 3).

4.1 The Basic Workflow

While our proposed workflow trains a network twice, it achieves universality - as we will show, it can be applied across a wide range of neural network architectures and tasks. It follows the basic train, prune, and fine-tune approach:

¹ <https://docs.nvidia.com/cuda/cusparselt/index.html>

² <https://docs.nvidia.com/cuda/cublas/index.html>

1. Train a model without sparsity,
2. Prune the model in a 2:4 sparse pattern,
3. Retrain the pruned model:
 - initialize the weights to the values from Step 2,
 - use the same optimizer and schedule (learning-rate, schedule, number of epochs, etc.) as in Step 1,
 - maintain the sparsity pattern computed in Step 2.

This workflow is implemented in the Automatic SParsity (ASP) [36] library for PyTorch and is illustrated in Figure 4, where the two training stages are identical in learning rate schedule and length and are separated by the one-shot pruning step. While Step 1 is straightforward, some details about Steps 2 and 3 follow.

Step 2: Weight pruning: At its simplest, the pruning step removes the two smallest weights in each group of four to meet the 2:4 pattern requirement, as illustrated in Figure 5. We found using the magnitude criteria sufficient, but one could consider other metrics, such as output activation similarity or instantaneous gradients.

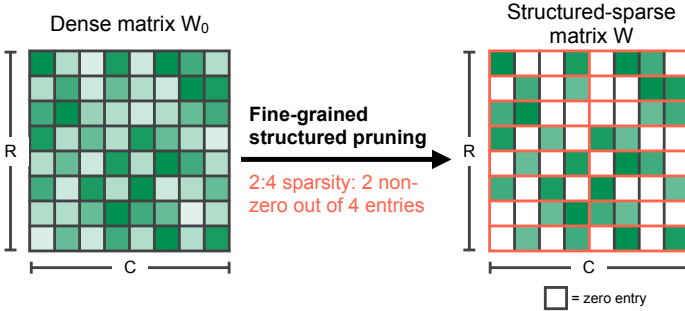


Fig. 5. Step 2 of the workflow modifies the weights in the dense matrix (W_0) on the left to conform to the 2:4 constraint, yielding the sparse matrix (W) on the right.

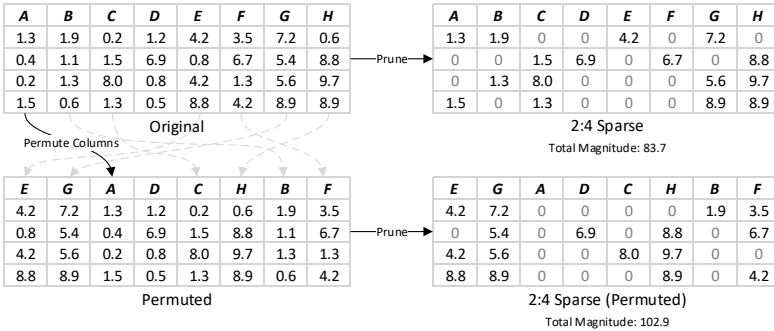


Fig. 6. Permuting columns of a weight matrix prior to pruning the matrix can reduce the effect of the 2:4 sparsity constraint on weight magnitude.

At this point in the workflow, the opportunity exists to change the layout of the network’s weights using channel permutations to minimize the impact of the pruning step. Figure 6 shows how this works on a random dense matrix (top-left). When this matrix is pruned with the 2:4 sparse constraint (top-right), some relatively large values are lost, resulting in a final total weight magnitude of 83.7. By first permuting columns

of this weight matrix to distribute the large values more evenly (lower-left), they are preserved after pruning (lower-right), for a final total weight magnitude of 102.9.

Since permutations are applied to columns of weights (the A matrix in Figure 2), the rows of activations (B) must be similarly permuted to maintain the same result for the GEMM operation. We accomplish this by permuting the *rows* of the weights used to produce those activations, typically the weights of the previous layer. For convolutions, permutations are applied to the input channel dimension, which becomes a component of the weight matrix’s column dimension. This process does not change the computations performed or the network’s results and incurs no runtime overhead.

For the majority of networks tested, these permutations are not required - step 2 is as simple as enforcing the 2:4 constraint on the weight tensors as they are. However, some networks are designed with efficiency in mind and begin with very few parameters; simply pruning and fine-tuning these models may still result in accuracy loss. Channel permutations make the most of each nonzero parameter and allow 2:4 sparsity to maintain accuracy for these parameter-efficient networks, as shown in Table 3. Details about finding quality permutations will be presented separately.

Step 3: Sparse retraining: We use retraining to recover model accuracy lost when half of the weights are removed by Step 2 of the workflow. To avoid any hyper-parameter search, we simply repeat the training session from Step 1, starting with weights from Step 1 rather than a random initialization. It is important to reset all hyper-parameters and optimizer state, such as momenta, etc. Any weight removed in Step 2 should retain its zero value in order to maintain the 2:4 sparsity pattern.

4.2 Layers to Prune

In our studies, we prune only layers that have learnable parameters and lead to a GEMM-like operation during the neural network’s execution on hardware. Such layers include convolution, fully-connected, and recurrent layers. We do not prune layers with inner dimensions (GEMM-K for fully-connected or recurrent layers and $C \times R \times S$ for convolution layers) that are not multiples of 16 and 32 for 16-bit floating point and 8b-integer formats, respectively. We also do not prune embedding layers (typical in language processing tasks) since these layers effectively implement a lookup table. Further, since our goal is to speed up inference, we do not prune layers that are involved only in the training phase and not in the inference phase of the network. Such layers include language-modeling heads used during training in language processing networks (like BERT [37]) which are then replaced by task-specific heads during inference, auxiliary classifiers used in Inception networks [38] which are removed altogether for deployment, and the entirety of discriminator networks used in adversarial training of generative networks (GANs).

4.3 Applying the Workflow to Models Trained in Multiple Phases

For models that are trained in a single phase, the application of the workflow from Section 4.1 is straightforward. Examples of single-phase training include image classification networks trained on the ILSVRC2012 [39] dataset, language translation networks trained on a single dataset, etc. However, when networks are trained in multiple phases, we can consider how many and which phases to consider for Steps 1 and 3 of the workflow. For example, object detection networks are often trained in 2 phases: first the backbone is trained on ILSVRC2012, then the detector heads are added and the model is trained for detection on COCO [40] dataset. Another example is question answering networks, such as BERT, that are first trained for language modeling and *then* trained for question answering on another dataset. We break such scenarios down into two categories:

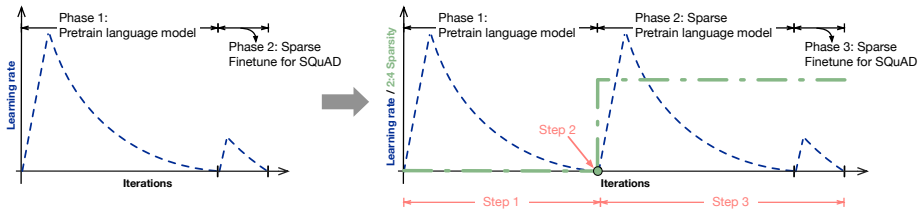


Fig. 7. BERT’s downstream tasks are too small to recover accuracy lost by pruning the language model, so the pre-training is repeated with sparsity before fine-tuning for the downstream task.

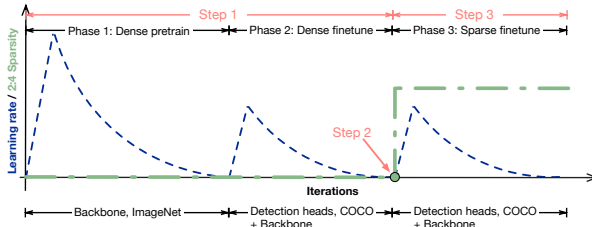


Fig. 8. A detection network’s fine-tuning of the detection heads is enough to recover accuracy lost by pruning the backbone, so the pruning step is inserted after the full network is trained, and only the fine-tuning is repeated.

Cases that require pruning and retraining the first phase: In some cases where the second phase trains on a small data set, we have observed that the pruned network does not go through enough updates to recover accuracy after pruning if only the second phase is used for the retraining step of the workflow. An example of this case is a language model like BERT, which is pre-trained on a very large dataset and then fine-tuned on a much smaller dataset for downstream tasks. The solution to this problem is to simply prune and retrain after the *pre-training* step, as shown in Figure 7. Then, the fine-tuning for the downstream task starts with a sparse and retrained model and simply maintains the sparsity pattern.

Cases that can prune and retrain only the second phase: In contrast to language models, common object detection tasks fine-tune with a large-enough data set to *not* require repeating the backbone’s pre-training. In this case, it is sufficient to train a dense backbone and fine-tune detection or segmentation heads. At this point, after all the weights have been trained, they can be pruned and the fine-tuning repeated, as shown in Figure 8. It is important to note that the task-specific heads have to be fine-tuned before they can be pruned.

These examples show different types of two-phase training; the same principles can be applied to training sessions consisting of more than two phases.

4.4 Combining Sparsity and Quantization

Quantization is a popular technique to accelerate neural network inference – by adapting the network to use narrower integer types, for example INT8, we can both reduce memory bandwidth pressure and benefit from higher throughput math pipelines [41]. Quantizing a network typically starts with a network trained in floating point, then calibration is applied to determine the parameters for replacing floating point values and math with low-bit integer ones. While some networks retain accuracy immediately after quantization, others require fine-tuning to recover lost accuracy. For both types of

networks our recommendation is to apply quantization calibration (and potentially fine-tuning) *after* a network has been pruned (and retrained) for sparsity.

5 Results

We evaluate the workflow proposed in Section 4 across a range of problem domains, tasks, and neural network architectures. For training each of the networks, we use hyper-parameters and training details mentioned in the papers introducing the network architecture and/or popular public repositories of network implementations. We examine model accuracy for both floating point networks as well as their quantization to INT8.

Table 2. Top-1 accuracy of image classification networks evaluated on the ImageNet ILSVRC2012 dataset with 2:4 sparsity.

Network	Accuracy		
	Dense FP16	Sparse FP16	Sparse INT8
ResNet-34	73.7	73.9	73.7
ResNet-50	76.1	76.2	76.2
ResNet-50 (SWSL)	81.1	80.9	80.9
ResNet-101	77.7	78.0	77.9
ResNeXt-50-32x4	77.6	77.7	77.7
ResNeXt-101-32x16	79.7	79.9	79.9
ResNeXt-101-32x16 (WSL)	84.2	84.0	84.2
DenseNet-121	75.5	75.3	75.3
DenseNet-161	78.8	78.8	78.9
Wide ResNet-50	78.5	78.6	78.5
Wide ResNet-101	78.9	79.2	79.1
Inception v3	77.1	77.1	77.1
Xception	79.2	79.2	79.2
VGG-11	70.9	70.9	70.8
VGG-16	74.0	74.1	74.1
VGG-19	75.0	75.0	75.0
SUNet-128	75.6	76.0	75.4
SUNet-7-128	76.4	76.5	76.3
DRN26	75.2	75.3	75.3
DRN-105	79.4	79.5	79.4

5.1 Image Classification Networks

Image classification networks are trained in a single phase, thus retraining simply repeats the training step schedule (with the exact same hyper-parameters and learning rate schedule as used to train the network) starting with the network initialized to its pruned trained weights.

Table 2 shows the accuracy of a wide variety of networks: popular networks like ResNet [42], VGG [43] and Inception [38], stacked U-Nets (SUNet) [44], dilated residual

Table 3. Small, parameter-efficient networks that struggle with the baseline fine-tuning approach match the target accuracy when permuting before fine-tuning (ILSVRC2012).

Network	Accuracy (FP16)		
	Dense	2:4 Sparse	
		Default	Permuted
MobileNet v2	71.55	69.56	71.56
SqueezeNet v1.0	58.09	54.08	58.38
SqueezeNet v1.1	58.21	56.96	58.23
MNASNet 1.0	73.24	71.99	73.27
ShuffleNet v2	68.32	66.97	68.42
EfficientNet B0	77.25	75.98	77.29
EfficientNet-WideSE B0	77.63	76.64	77.63

networks (DRN) [45]. We also examine networks trained with weakly (WSL) or semi-weakly supervised learning (SWSL) methods [46, 47] which use additional data to improve accuracy. We prune all the convolution and fully-connected layers except for the first one (7×7 convolution on 3-channel input) since they do not have a GEMM-K dimension that is an even multiple of 16.

Weight and input activation tensors in convolution layers, including the first layer, and fully-connected layers are quantized to INT8. Entropy and max calibration [41] are used for activations and weights, respectively. Per-tensor scaling factors are used for activation tensors, per-channel scaling factors are used for weight tensors in convolutions, and per-row scaling factors are used for weight tensors in fully-connected layers.

The results in Table 2 indicate the accuracy is maintained for both floating point and quantized networks when sparsity workflow from Section 4 is applied. While for some networks, sparse and quantized models accuracy is slightly different than for the dense non-quantized counterparts, these differences are within bounds of run-to-run variation caused by random seeds or fine-tuning non-determinism.

Some of the lower-parameter networks (MobileNet v2 [48], SqueezeNet [49], MNASNet [50], ShuffleNet v2 [51], and EfficientNet [52]) do not fully recover accuracy when applying the basic workflow. As Table 3 shows, permuting the weights before pruning allows a fully recovery of accuracy for these models.

5.2 Image Segmentation and Detection Networks

To study object detection and segmentation, we use networks from PyTorch Torchvision, Detectron2 [53], NVIDIA Deep Learning Examples for Tensor Cores [54], and NVIDIA ADLR [55] repositories.

Image segmentation and object detection models are typically trained in two phases: first a backbone is trained for image classification, followed by the addition of model components (segmentation/detection heads, FPN, etc.), and then training for detection or segmentation. Backbones are trained on the ILSVRC2012 dataset, downstream tasks are trained on COCO 2017 [40], with some semantic segmentation networks also using Mapillary and Cityscapes datasets.

Since these detection and segmentation datasets are relatively large, we find that we can prune the weights after the second phase and repeat the training of only the second phase. Accuracy results are summarized in Table 4, which shows sparse results matching those of the dense counterparts.

Table 4. Accuracy of detection (bbox Average Precision, top) and image segmentation (mask Average Precision, bottom) networks with 2:4 sparsity on the COCO 2017 dataset. In this table, RN=ResNet, FPN=Feature Pyramid Network, and RPN=Region Proposal Network. 1x and 3x are the length of fine-tuning schedules as referred to in Detectron2.

Network	Accuracy		
	Dense FP16	Sparse FP16	Sparse INT8
MaskRCNN-RN50	37.9	37.9	37.8
RetinaNet-RN50	34.8	35.2	35.1
SSD-RN50	24.8	24.8	24.9
FasterRCNN-RN50-FPN-1x	37.6	38.6	38.4
FasterRCNN-RN50-FPN-3x	39.8	39.9	39.4
FasterRCNN-RN101-FPN-3x	41.9	42.0	41.8
MaskRCNN-RN50-FPN-1x	39.9	40.3	40.0
MaskRCNN-RN50-FPN-3x	40.6	40.7	40.4
MaskRCNN-RN101-FPN-3x	42.9	43.2	42.8
RetinaNet-RN50-FPN-1x	36.4	37.4	37.2
RPN-RN50-FPN-1x	45.8	45.6	45.5
MaskRCNN-RN50	34.5	34.6	34.5
MaskRCNN-RN50-FPN-3x	36.9	37.0	-
MaskRCNN-RN101-FPN-3x	38.7	38.9	-
ResNeXt-101*	81.4	81.7	81.2

*IoU of network pre-trained on Mapillary and fine-tuned on Cityscapes dataset

For each network, all layers in the backbone (except the very first 3-channel convolution) and heads are pruned with 2:4 sparsity, and all layers (including the first convolution) are quantized to INT8. Similar to classification networks, entropy calibration with per-tensor scaling factors are used for activation tensors, and max calibration with per-channel or per-row scaling factors are used for weight tensors.

5.3 Generative Adversarial Networks (GANs)

As part of devising a scheme for stabilizing the fine-tuning of sparse GANs [56], we apply the procedure from Section 4 to generate 2:4 sparse floating-point networks to a variety of GANs and tasks. The results for Frechet Inception Distance (FID) scores [57] (lower is better) from this work are shown in Table 5.

5.4 Networks for Natural Language Processing (NLP)

To study the behavior of our workflow on NLP tasks, we select recurrent-based translation network (GNMT [58]), Transformer-based translation network (FairSeq Transformer [59]), and two language modeling networks (Transformer-XL [60] and BERT [37]).

Language Translation: Language translation networks are trained in a single-phase, thus the training session is repeated using the original hyper-parameters after pruning. After fine-tuning, the networks are quantized to INT8 using max calibration [41]. Per-tensor scaling factors are used for activation tensors and per-row scaling factors are used

Table 5. FID scores (lower is better) of GANs with 2:4 sparsity.

Task	Network	Dataset	FID score	
			Dense	2:4 Sparse
Image Synthesis	DCGAN	MNIST	50.39	50.54
Domain Translation	Pix2Pix	Sat \rightarrow Map	17.64	17.89
Domain Translation	Pix2Pix	Sat \leftarrow Map	30.83	30.72
Style Transfer	CycleGAN	Monet \rightarrow Photo	63.15	63.00
Style Transfer	CycleGAN	Monet \leftarrow Photo	31.99	32.36
Image-Image Translation	CycleGAN	Zebra \rightarrow Horse	60.93	61.03
Image-Image Translation	CycleGAN	Zebra \leftarrow Horse	52.86	52.45
Image-Image Translation	StarGAN	CelebA	6.11	6.93
Super Resolution	SRGAN	DIV2K	14.65	16.60

Table 6. Accuracy on En-De WMT’14 of two network architectures for language translation. BLEU score is reported for accuracy in the table.

Network	Accuracy		
	Dense FP16	Sparse FP16	Sparse INT8
GNMT	24.6	24.9	24.9
FairSeq Transformer	28.2	28.5	28.3

for weight tensors in fully-connected and recurrent layers. Table 6 shows the accuracy of networks retrained for the 2:4 pattern matching that of the dense originals.

Language Modeling: The Enwik8 [61] dataset is used for Transformer-XL evaluation, and SQuAD v1.1 [62] is used for BERT evaluation. State-of-art language modeling networks involve a two-stage training process: unsupervised training (also called pre-training) on large-scale unlabeled data sets, followed by fine-tuning on much smaller data sets for downstream tasks, such as question-answering and entailment classification. For these networks, our studies show that repeating pre-training step after pruning and then fine-tuning the sparse network on target task leads to a model that matches dense network’s accuracy (shown in Table 7). We use the BERT_{LARGE} model and training scripts as described in NVIDIA Megatron repository [63]. For both networks, all GEMM layers involving weight/parameter tensors inside a transformer block/layer are pruned (attention layers are not pruned since they do not involve any weights).

When quantizing BERT to INT8, all GEMM layers including batched-GEMM layers operate on INT8 operands, along with data in the residual connections. This aggressive quantization can cause a small accuracy degradation, but sparsity matches the dense

Table 7. Accuracy of dense and sparse 12-layer Transformer-XL models in bits per character (BPC) on enwik8 and BERT_{LARGE} F1 score on SQuAD v1.1.

Network	Metric	Accuracy			
		Dense FP16	Sparse FP16	Dense INT8	Sparse INT8
12-layer Transformer-XL	BPC	1.06	1.06	-	-
BERT _{LARGE}	F1	91.9	91.9	90.9	90.8

accuracy in both cases. Per-tensors scaling factors are used with max calibration for weight tensors and percentile calibration [41] for activations tensors in fully-connected layers. The pruned pre-trained language model is fine-tuned for both quantization operations as well as the SQuAD dataset’s task-specific heads at the same time.

6 Conclusions and Future Work

Sparsity in neural networks remains an active research area. Unstructured sparsity, which is the subject of many research efforts, requires very high levels of sparsity in order to achieve speedups over dense math on modern processors with matrix-math pipelines. However, very sparse networks have difficulty maintaining model accuracy. To overcome these challenges, we introduced 2:4 structured sparsity, hardware primitives for its acceleration, and a workflow for pruning networks. The NVIDIA Ampere GPU architecture introduces Sparse Tensor Cores, which have $2\times$ math throughput for GEMM-like operations (convolutions and matrix multiplies) where the first argument is a tensor with 2:4 sparsity. The workflow was empirically shown to maintain accuracy over a wide range of tasks and neural network models, trained using standard learning rate schedules found in public code repositories.

The proposed sparsity workflow repeats a training session after pruning the weights of a trained dense networks. The benefit of this approach is that retraining does not require any hyper-parameter changes or searches. However, since this workflow doubles the training time, an interesting direction for future investigations is finding shorter fine-tuning schedules. Some preliminary experiments with a grid-search have identified a set of hyper-parameters (initial learning rate, learning rate schedule and epochs/iterations to fine-tune) for shorter fine-tuning sessions that maintained accuracy. However, these parameters were highly network- and task-dependent, and we have not yet been able to identify hyper-parameters that work universally for various tasks and networks. Thus, a universal approach to reduced fine-tuning requirements remains future work. Another interesting direction for future work is exploring the effects of sparsity on models that were trained to the limits of their capacity - there are indications that some popular models can achieve higher accuracy when trained on larger datasets or with longer training schedules. While we looked at some models trained on larger datasets (with weakly and semi-weakly supervised techniques) in Section 5, evaluating alternative training schedules is an interesting next step.

Our proposed workflow targets acceleration of inference. While this matches what would be needed to accelerate the forward pass of training, in order to also accelerate the backward pass of training, the 2:4 constraint must also be satisfied by the transposed weight tensors. This can be done by enforcing the constraint along both dimensions of the weight matrix: rows *and* columns. The PyTorch ASP [36] library provides a simple greedy approach, as well as an exhaustive search, that seeks to minimize the weight magnitude lost by pruning; investigation of more efficient mask-finding algorithms is an active research area (for example, [64]) and is left as future work. To accelerate training, one would also aim to minimize the number of updates performed with a dense network. For this, one may need to train with dynamic sparsity masks, evolving them during training [65] (as opposed to static masks computed once during pruning, which this paper shows to suffice for inference). Thus, investigation of dynamic mask requirements is an intriguing area for future work as well.

Finally, it is also interesting to investigate pruning of activations, as some layers, such as multi-head attention in Transformer-based networks, do not involve any weights. Our preliminary experiments suggest that it is possible to prune activations with a 2:4 pattern without accuracy loss; a fully general methodology is future work.

References

1. Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017.
2. Joel Hestness, Newsha Ardalani, and Greg Diamos. Beyond human-level accuracy: Computational challenges in deep learning. *CoRR*, abs/1909.01736, 2019.
3. Md. Mostofa Ali Patwary, Milind Chabbi, Heewoo Jun, Jiaji Huang, Gregory Frederick Diamos, and Kenneth Church. Language modeling at scale. *CoRR*, abs/1810.10045, 2018.
4. Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018.
5. Nikko Ström. Sparse connection and pruning in large dynamic artificial neural networks. In *EUROSPEECH*, 1997.
6. Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Shijian Tang, Erich Elsen, Bryan Catanzaro, John Tran, and William J. Dally. DSD: regularizing deep neural networks with dense-sparse-dense training flow. In *ICLR*, 2017.
7. Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, pages 1135–1143, 2015.
8. Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the granularity of sparsity in convolutional neural networks. In *CVPR Workshops*, pages 13–20, 2017.
9. Michael Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *ICLR Workshops*, 2018.
10. Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *CoRR*, abs/1707.06168, 2017.
11. Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.
12. Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018.
13. Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.
14. NVIDIA A100 tensor core GPU architecture. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf>, 2020.
15. Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *CVPR*, 2019.
16. Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: single-shot network pruning based on connection sensitivity. In *ICLR*, 2019.
17. Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient DNN inference on GPU. *CoRR*, abs/1811.00206, 2018.
18. Huan Wang, Qiming Zhang, Yuehai Wang, and Haoji Hu. Structured pruning for efficient convnets via incremental regularization. *CoRR*, abs/1811.08390, 2018.
19. Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *CoRR*, abs/1512.08571, 2015.
20. Mingbao Lin, Rongrong Ji, Shaojie Li, Qixiang Ye, Yonghong Tian, Jianzhuang Liu, and Qi Tian. Filter sketch for network pruning. *CoRR*, abs/2001.08514, 2020.
21. Noah Gamboa, Kais Kudrolli, Anand Dhoot, and Ardavan Pedram. Campfire: Compressible, regularization-free, structured sparse training for hardware accelerators. *CoRR*, abs/2001.03253, 2020.

22. Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *MICRO*, 2019.
23. Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016.
24. Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. The lottery ticket hypothesis at scale. *CoRR*, abs/1903.01611, 2019.
25. Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing fine-tuning and rewinding in neural network pruning. In *ICLR*, 2020.
26. Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *CoRR*, abs/1704.05119, 2017.
27. Sharan Narang, Eric Undersander, and Gregory F. Diamos. Block-sparse recurrent neural networks. *CoRR*, abs/1711.02782, 2017.
28. Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. *CoRR*, abs/1911.09723, 2019.
29. Mi Sun Park, Xiaofan Xu, and Cormac Brick. Squantizer: Simultaneous learning for both sparse and low-precision neural networks. *CoRR*, abs/1812.08301, 2018.
30. Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aäron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. *CoRR*, abs/1802.08435, 2018.
31. Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. *CoRR*, abs/1911.11134, 2019.
32. Scott Gray, Alec Radford, and Diederik P. Kingma. *GPU Kernels for Block-Sparse Weights*, Accessed April 13, 2021. <https://openai.com/blog/block-sparse-gpu-kernels/>.
33. H. Kang. Accelerator-aware pruning for convolutional neural networks. *TCSVT*, 30(7):2093-2103, 2020.
34. Sparse matrix storage formats. <https://www.gnu.org/software/gsl/doc/html/spmatrix.html>.
35. Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.
36. NVIDIA’s Automatic SParsity (ASP) library. <https://github.com/NVIDIA/apex/tree/master/apex/contrib/sparsity>, 2020.
37. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
38. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.
39. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211-252, 2015.
40. Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.
41. Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *CoRR*, abs/2004.09602, 2020.

42. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
43. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
44. Sohil Shah, Pallabi Ghosh, Larry S. Davis, and Tom Goldstein. Stacked u-nets: A no-frills approach to natural image segmentation. *CoRR*, abs/1804.10343, 2018.
45. Fisher Yu, Vladlen Koltun, and Thomas A. Funkhouser. Dilated residual networks. *CoRR*, abs/1705.09914, 2017.
46. Dhruv Kumar Mahajan, Ross B. Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. In *ECCV*, 2018.
47. I. Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. Billion-scale semi-supervised learning for image classification. *CoRR*, abs/1905.00546, 2019.
48. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
49. Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.
50. Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.
51. Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.
52. Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.
53. Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. <https://github.com/facebookresearch/detectron2>, 2019.
54. Nvidia deep learning examples. <https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Detection/SSD>.
55. Yi Zhu, Karan Sapra, Faisal A. Reda, Kevin J. Shih, Shawn Newsam, Andrew Tao, and Bryan Catanzaro. Improving semantic segmentation via video propagation and label relaxation. In *CVPR*, 2019.
56. Chong Yu and Jeff Pool. Self-supervised generative adversarial compression. In *NeurIPS*, 2020.
57. Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local nash equilibrium. In *NeurIPS*, 2017.
58. Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.
59. Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL-HLT: Demonstrations*, 2019.
60. Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019.
61. *The Hutter Prize Wikipedia dataset*, Accessed April 13, 2020. <http://prize.hutter1.net/>.

62. Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.
63. *Megatron-LM repo*, Accessed April 13, 2020. <https://github.com/NVIDIA/Megatron-LM>.
64. Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Seffi Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find $n:m$ transposable masks. *CoRR*, abs/2102.08124, 2021.
65. Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N:M fine-grained structured sparse neural networks from scratch. In *ICLR*, 2021.