ACCELERATING SPARSITY IN THE NVIDIA AMPERE ARCHITECTURE

Jeff Pool, Senior Architect
OUTLINE

Sparsity Review
Motivation
Taxonomy
Challenges

NVIDIA A100 GPU 2:4 Sparsity
Sparsity pattern
Sparse Tensor Cores
Inference Speedups

Training Recipe
Recipe steps
Empirical evaluation
Implementation in frameworks
SPARSITY - INFERENCE ACCELERATION
VS TRAINING ACCELERATION

Focus of this talk is Inference acceleration

• Including training methods that enable accelerated inferencing with no loss of accuracy

Using sparsity to accelerate training is very interesting - but not the focus of this talk!
• At the end of the talk, we’ll touch briefly on accelerating training
SPARSITY: ONE OF MANY OPTIMIZATION TECHNIQUES

Optimization goals for inference:

• Reduce network model size
• Speed up network model execution

Observations that inspire sparsity investigations

• Biology: neurons are not densely connected
• Neural networks:
  • Trained model weights have many small-magnitude values
  • Activations may have 0s because of ReLU

Figure: “DSD: Dense-Sparse-Dense Training for Deep Neural Networks” S. Han et al.
SPARSITY AND PERFORMANCE

Do not store or process 0 values -> smaller and hopefully faster model

- Eliminate (prune) connections: set some weights to 0
- Eliminate (prune) neurons
- Etc.

But, must also:

- Maintain model accuracy
- Efficiently execute on hardware to gain speedup
PRUNING/SPARSITY IS AN ACTIVE RESEARCH AREA

Optimal Brain Damage

Yann Le Cun, John S. Denker and Sara A. Soila
AT&T Bell Laboratories, Holmdel, N. J. 07733

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilys@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca
SPARSITY TAXONOMY

Structure:
- Unstructured: irregular, no pattern of zeros
- Structured: regular, fixed set of patterns to choose from

Granularity:
- Finest: prune individual values
- Coarser: prune blocks of values
- Coarsest: prune entire layers
STATE OF SPARSITY RESEARCH

Lots of research in two areas:

- High amounts (80-95%) unstructured, fine-grained sparsity
- Coarse-grained sparsity for simpler acceleration

Challenges not resolved for these approaches:

- **Accuracy loss**
  - High sparsity often leads to accuracy loss of a few percentage points, even after advanced training techniques

- **Absence of a training approach that works across different tasks and networks**
  - Training approaches to recover accuracy vary from network to network, often require hyper-parameter searches

- **Lack of speedup**
  - Math: unstructured data struggles to take advantage of modern vector/matrix math instructions
  - Memory access: unstructured data tends to poorly utilize memory buses, increases latency due to dependent sequences of reads
  - Storage overheads: metadata can consume 2x more storage than non-zero weights, undoing some of compression benefits
SPARSITY SUPPORT INTRODUCED IN NVIDIA AMPERE ARCHITECTURE
SPARSITY IN A100 GPU

Fine-grained structured sparsity for Tensor Cores

- 50% fine-grained sparsity
- 2:4 pattern: 2 values out of each contiguous block of 4 must be 0

Addresses the 3 challenges:

- **Accuracy**: maintains accuracy of the original, unpruned network
  - Medium sparsity level (50%), fine-grained
- **Training**: a recipe shown to work across tasks and networks
- **Speedup**:
  - Specialized Tensor Core support for sparse math
  - Structured: lends itself to efficient memory utilization

2:4 structured-sparse matrix

= zero value
SPARSE TENSOR CORES

Applicable for:

- Convolutions
- Matrix multiplies (linear layers, MLPs, recurrent cells, transformer blocks, etc.)

Inputs: sparse weights, dense activations

Output: dense activations

Compressed format for the sparse matrix:

- Do not store two 0s in each block of 4 values -> 50% of original storage
  - If a block contains more than two 0s, some of the 0s will be stored
- Metadata to index the remaining 2 values - needed for accessing the dense activations
  - 2 bits per value
  - 12.5% overhead for fp16, compared to 100-200% for CSR format
2:4 COMPRESSED MATRIX FORMAT

At most 2 non-zeros in every contiguous group of 4 values

Compressed Matrix:

Data: ½ size

Metadata: 2b per non-zero element

16b data => 12.5% overhead
8b data => 25% overhead
TENSOR CORE OPERATION

Tiling a Large GEMM

Dense Tensor Cores (FP16)

16x16 * 16x8 matrix multiplication

Replicated and repeated to support large M, N, K
Dense Tensor Cores (FP16)

$16 \times 32 \times 32 \times 8$ matrix multiplication - 2 cycles

Larger Tile = More Cycles
TENSOR CORE OPERATION

Pruned Weight Matrix

A: Sparse, MxK
B: Dense, KxN
C: Dense, MxN

16x32
32x8
16x8
16x8
TENSOR CORE OPERATION

Pruned and Compressed Weight Matrix

A: Sparse, MxK

B: Dense, KxN

C: Dense, MxN

16x32

32x8

16x8
TENSOR CORE OPERATION

Tiling a Large, Sparse GEMM

A: Sparse, MxK/2

16x16 32

Compressed!

B: Dense, KxN

32x8

C: Dense, MxN

16x8

Fine-grained structured-sparse matrix format
TENSOR CORE OPERATION

Sparse Tensor Cores - Hardware Magic

A: Sparse, MxK/2

B: Dense, KxN

16x16

32x8

Compressed!

C: Dense, MxN

16x8
Sparse Tensor Cores (FP16)

16x32 * 32x8 effective matrix multiplication - 1 cycle

2x the work with the same instruction throughput
**TENSOR CORE MATH THROUGHPUT**

2x with Sparsity

<table>
<thead>
<tr>
<th>INPUT OPERANDS</th>
<th>ACCUMULATOR</th>
<th>TOPS</th>
<th>Dense vs. FFMA</th>
<th>Sparse vs. FFMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>19.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TF32</td>
<td>FP32</td>
<td>156</td>
<td>8X</td>
<td>16X</td>
</tr>
<tr>
<td>FP16</td>
<td>FP32</td>
<td>312</td>
<td>16X</td>
<td>32X</td>
</tr>
<tr>
<td>BF16</td>
<td>FP32</td>
<td>312</td>
<td>16X</td>
<td>32X</td>
</tr>
<tr>
<td>FP16 INT8</td>
<td>FP16 INT32</td>
<td>312</td>
<td>16X</td>
<td>32X</td>
</tr>
<tr>
<td>INT8</td>
<td>INT32</td>
<td>624</td>
<td>32X</td>
<td>64X</td>
</tr>
<tr>
<td>INT4</td>
<td>INT32</td>
<td>1248</td>
<td>64X</td>
<td>128X</td>
</tr>
<tr>
<td>BINARY</td>
<td>INT32</td>
<td>4992</td>
<td>256X</td>
<td>-</td>
</tr>
</tbody>
</table>
SPARSE TENSOR CORES

Measured GEMM Performance with Current Software

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>K</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>8192</td>
<td>1024</td>
<td>1.44x</td>
</tr>
<tr>
<td>1024</td>
<td>16384</td>
<td>1024</td>
<td>1.73x</td>
</tr>
<tr>
<td>4096</td>
<td>8192</td>
<td>1024</td>
<td>1.53x</td>
</tr>
<tr>
<td>4096</td>
<td>16384</td>
<td>1024</td>
<td>1.78x</td>
</tr>
</tbody>
</table>

GEMM sizes selected from BERT-Large
# Sparse Tensor Cores

## Measured Convolution Performance With Current Software

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>K</th>
<th>H,W</th>
<th>R,S</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>2048</td>
<td>14</td>
<td>1</td>
<td>1.52x</td>
</tr>
<tr>
<td>32</td>
<td>2048</td>
<td>1024</td>
<td>14</td>
<td>1</td>
<td>1.77x</td>
</tr>
<tr>
<td>32</td>
<td>2048</td>
<td>4096</td>
<td>7</td>
<td>1</td>
<td>1.64x</td>
</tr>
<tr>
<td>32</td>
<td>4096</td>
<td>2048</td>
<td>7</td>
<td>1</td>
<td>1.75x</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>512</td>
<td>7</td>
<td>3</td>
<td>1.85x</td>
</tr>
</tbody>
</table>

Kernel sizes selected from ResNeXt-101_32x16d/ResNet-50
## NETWORK PERFORMANCE

### End to End Inference Speedup

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>DATA TYPE</th>
<th>SCENARIO</th>
<th>PERFORMANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Large</td>
<td>INT8</td>
<td>BS=256, SeqLen=128</td>
<td>6200 seq/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256, SeqLen=128</td>
<td>1.3X-1.5X</td>
</tr>
</tbody>
</table>
## NETWORK PERFORMANCE

### End to End Inference Speedup

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>DATA TYPE</th>
<th>SCENARIO</th>
<th>PERFORMANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Large</td>
<td>INT8</td>
<td>BS=256, SeqLen=128</td>
<td>6200 seq/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256, SeqLen=128</td>
<td>1.3X-1.5X</td>
</tr>
<tr>
<td>ResNeXt-101_32x16d</td>
<td>FP16</td>
<td>BS=256</td>
<td>2700 images/second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256</td>
<td>Up to 1.3X</td>
</tr>
</tbody>
</table>
## NETWORK PERFORMANCE

### End to End Inference Speedup

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>DATA TYPE</th>
<th>SCENARIO</th>
<th>PERFORMANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Large</td>
<td>INT8</td>
<td>BS=256, SeqLen=128</td>
<td>6200 seq/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256, SeqLen=128</td>
<td>1.3X-1.5X</td>
</tr>
<tr>
<td>ResNeXt-101_32x16d</td>
<td>FP16</td>
<td>BS=256</td>
<td>2700 images/second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256</td>
<td>Up to 1.3X</td>
</tr>
<tr>
<td></td>
<td>INT8</td>
<td>BS=256</td>
<td>4400 images/second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS=1-256</td>
<td>Up to 1.3X</td>
</tr>
</tbody>
</table>
1.8x GEMM Performance -> 1.5x Network Performance

Some operations remain dense:
Non-GEMM layers (Softmax, Residual add, Normalization, Activation functions, …)
GEMMs without weights to be pruned - Attention Batched Matrix Multiplies
CONVOLUTION SPEEDUPS

Layers of ResNeXt-101

Some layers are less compute-limited than others

![Graph showing speedup of INT8 ResNeXt-101_32x16d Convolutions](image-url)
TRAINING RECIPE
GOALS FOR A TRAINING RECIPE

Maintains accuracy

Is applicable across various tasks, network architectures, and optimizers

Does not require hyper-parameter searches
RECIPE FOR 2:4 SPARSE NETWORK TRAINING

1) Train (or obtain) a dense network

2) Prune for 2:4 sparsity

3) Repeat the original training procedure
   • Same hyper-parameters as in step-1
   • Initialize to weights from step-2
   • Maintain the 0 pattern from step-2: no need to recompute the mask
RECIPE STEP 2: PRUNE WEIGHTS

Single-shot, magnitude-based pruning

For each 1x4 block of weights:

- Set 2 weights with the smallest magnitudes to 0

Layer weights to prune: conv, linear
RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix $W$

Structured-sparse matrix $W$

Fine-grained structured pruning

2:4 sparsity: 2 non-zero out of 4 entries

$\square = \text{zero value}$
RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix $W$

Structured-sparse matrix $W$

Fine-grained structured pruning

2:4 sparsity: 2 non-zero out of 4 entries

$\square$ = zero value
RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix $W$

Structured-sparse matrix $W$

Fine-grained structured pruning

2:4 sparsity: 2 non-zero out of 4 entries

☐ = zero value
RECIPE STEP 3: RETRAIN

Pruning out 50% of the weight values reduces model accuracy

Retraining recovers accuracy

• Adjusts the remaining weights to compensate for pruning

• Requirement intuition:
  • Need **enough updates** by optimizer to compensate for pruning
  • Updates need **high-enough learning rates** to compensate

Simplest retraining:

• Repeat the training session, starting with weight values after pruning (as opposed to random initialization)

• All the same training hyper-parameters

• Do not update weights that were pruned out
EXAMPLE LEARNING RATE SCHEDULE
STEP 3 FOR NETWORKS TRAINED IN MULTIPLE PHASES

Some networks are trained in multiple phases

• Pretrain on one task and dataset, then train (fine-tune) on another task and dataset

• Examples:
  • Retinanet for object detection: 1) train for classification on ImageNet, 2) train for detection on COCO
  • BERT for question answering: 1) train for language modeling on BooksCorpus/Wikipedia, 2) train for question answering on SQuAD

In some cases Step 3 can be applied to only the last phase of original training

• Shortens retraining to recover accuracy

• Generally requires that the last phase(s):
  • Perform enough updates
  • Use datasets large enough to not cause overfitting

• When in doubt - retrain from the earliest phase, carry the sparsity through all the phases
STEP3: DETECTOR EXAMPLE

Detection Dataset is Large Enough to Provide Enough Updates and Not Overfit

Phase 1: Dense Pre-Train

Phase 2: Dense Fine-Tune

Sparse Retrain: Phase 2

Backbone, ImageNet

Detection Heads, COCO +Backbone

Detection Heads, COCO +Backbone

Step 1

Step 2

Step 3
STEP 3: BERT SQUAD EXAMPLE

Squad Dataset and Fine-tuning is Too Small to Compensate for Pruning on its Own

Phase 1: Pretrain language model
Phase 2: Finetune for SQuAD

Phase 1: Sparse Pretrain language model
Phase 2: Sparse Finetune for SQuAD

Learning Rate

Step 1
Step 2
Step 3
SPARSITY AND QUANTIZATION

Apply Sparsity Before Quantizing

- Quantization
  - Generate a floating-point network
  - Apply quantization (calibration, fine-tuning)

- Quantization+Sparsity
  - Generate a floating-point network
  - Prune
  - Apply quantization (calibration, fine-tuning)
SPARSITY AND QUANTIZATION

Post-Training Quantization

Post-training calibration follows the sparse fine-tuning
SPARSITY AND QUANTIZATION

Quantization Aware Training

Fine-tune for sparsity before fine-tuning for quantization

- S22075: Integer Quantization for DNN Inference Acceleration
ACCURACY EVALUATION
ACCURACY

Overview

Tested 34 networks, covering a variety of AI domains, with the described recipe

- Run one test without sparsity and one test with sparsity, compare results

Results: accuracy is ~same (within prior observed run-to-run variation of networks)

FP16 networks trained with mixed precision training

INT8 networks generated by:

1\textsuperscript{st}: Retrain a sparse FP16 network first

2\textsuperscript{nd}: Apply traditional quantization techniques:

- Post-training calibration
- Quantization-Aware fine-tuning
## IMAGE CLASSIFICATION

### ImageNet

<table>
<thead>
<tr>
<th>Network</th>
<th>Dense FP16</th>
<th>Sparse FP16</th>
<th>Sparse INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-34</td>
<td>73.7</td>
<td>73.9</td>
<td>73.7</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>76.6</td>
<td>76.8</td>
<td>76.8</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>77.7</td>
<td>78.0</td>
<td>77.9</td>
</tr>
<tr>
<td>ResNeXt-50-32x4d</td>
<td>77.6</td>
<td>77.7</td>
<td>77.7</td>
</tr>
<tr>
<td>ResNeXt-101-32x16d</td>
<td>79.7</td>
<td>79.9</td>
<td>79.9</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>75.5</td>
<td>75.3</td>
<td>75.3</td>
</tr>
<tr>
<td>DenseNet-161</td>
<td>78.8</td>
<td>78.8</td>
<td>78.9</td>
</tr>
<tr>
<td>Wide ResNet-50</td>
<td>78.5</td>
<td>78.6</td>
<td>78.5</td>
</tr>
<tr>
<td>Wide ResNet-101</td>
<td>78.9</td>
<td>79.2</td>
<td>79.1</td>
</tr>
<tr>
<td>Inception v3</td>
<td>77.1</td>
<td>77.1</td>
<td>77.1</td>
</tr>
<tr>
<td>Xception</td>
<td>79.2</td>
<td>79.2</td>
<td>79.2</td>
</tr>
<tr>
<td>VGG-16</td>
<td>74.0</td>
<td>74.1</td>
<td>74.1</td>
</tr>
<tr>
<td>VGG-19</td>
<td>75.0</td>
<td>75.0</td>
<td>75.0</td>
</tr>
</tbody>
</table>
## IMAGE CLASSIFICATION

**ImageNet**

<table>
<thead>
<tr>
<th>Network</th>
<th>Dense FP16</th>
<th>Sparse FP16</th>
<th>Sparse INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50 (SWSL)</td>
<td>81.1</td>
<td>80.9</td>
<td>80.9</td>
</tr>
<tr>
<td>ResNeXt-101-32x8d (SWSL)</td>
<td>84.3</td>
<td>84.1</td>
<td>83.9</td>
</tr>
<tr>
<td>ResNeXt-101-32x16d (WSL)</td>
<td>84.2</td>
<td>84.0</td>
<td>84.2</td>
</tr>
<tr>
<td>SUNet-7-128</td>
<td>76.4</td>
<td>76.5</td>
<td>76.3</td>
</tr>
<tr>
<td>DRN-105</td>
<td>79.4</td>
<td>79.5</td>
<td>79.4</td>
</tr>
</tbody>
</table>
## SEGMENTATION/DETECTION

COCO 2017, bbox AP

<table>
<thead>
<tr>
<th>Network</th>
<th>Dense FP16</th>
<th>Sparse FP16</th>
<th>Sparse INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskRCNN-RN50</td>
<td>37.9</td>
<td>37.9</td>
<td>37.8</td>
</tr>
<tr>
<td>SSD-RN50</td>
<td>24.8</td>
<td>24.8</td>
<td>24.9</td>
</tr>
<tr>
<td>FasterRCNN-RN50-FPN-1x</td>
<td>37.6</td>
<td>38.6</td>
<td>38.4</td>
</tr>
<tr>
<td>FasterRCNN-RN50-FPN-3x</td>
<td>39.8</td>
<td>39.9</td>
<td>39.4</td>
</tr>
<tr>
<td>FasterRCNN-RN101-FPN-3x</td>
<td>41.9</td>
<td>42.0</td>
<td>41.8</td>
</tr>
<tr>
<td>MaskRCNN-RN50-FPN-1x</td>
<td>39.9</td>
<td>40.3</td>
<td>40.0</td>
</tr>
<tr>
<td>MaskRCNN-RN50-FPN-3x</td>
<td>40.6</td>
<td>40.7</td>
<td>40.4</td>
</tr>
<tr>
<td>MaskRCNN-RN101-FPN-3x</td>
<td>42.9</td>
<td>43.2</td>
<td>42.8</td>
</tr>
<tr>
<td>RetinaNet-RN50-FPN-1x</td>
<td>36.4</td>
<td>37.4</td>
<td>37.2</td>
</tr>
<tr>
<td>RPN-RN50-FPN-1x</td>
<td>45.8</td>
<td>45.6</td>
<td>45.5</td>
</tr>
</tbody>
</table>

RN = ResNet Backbone
FPN = Feature Pyramid Network
RPN = Region Proposal Network
## NLP - TRANSLATION

### EN-DE WMT’14

<table>
<thead>
<tr>
<th>Network</th>
<th>Metric</th>
<th>Dense FP16</th>
<th>Sparse FP16</th>
<th>Sparse INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNMT</td>
<td>BLEU</td>
<td>24.6</td>
<td>24.9</td>
<td>0.3</td>
</tr>
<tr>
<td>FairSeq Transformer</td>
<td>BLEU</td>
<td>28.2</td>
<td>28.5</td>
<td>0.3</td>
</tr>
<tr>
<td>Levenstein Transformer</td>
<td>Validation Loss</td>
<td>6.16</td>
<td>6.18</td>
<td>-0.2</td>
</tr>
</tbody>
</table>
## NLP - LANGUAGE MODELING

Transformer-XL, BERT

<table>
<thead>
<tr>
<th>Network</th>
<th>Task</th>
<th>Metric</th>
<th>Dense FP16</th>
<th>Sparse FP16</th>
<th>Sparse INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer-XL</td>
<td>enwik8</td>
<td>BPC</td>
<td>1.06</td>
<td>1.06</td>
<td>-</td>
</tr>
<tr>
<td>BERT-Base</td>
<td>SQuAD v1.1</td>
<td>F1</td>
<td>87.6</td>
<td>88.1</td>
<td>0.5</td>
</tr>
<tr>
<td>BERT-Large</td>
<td>SQuAD v1.1</td>
<td>F1</td>
<td>91.1</td>
<td>91.5</td>
<td>0.4</td>
</tr>
</tbody>
</table>
COMPARING 2:4 TO OTHER ALTERNATIVES

Alternatives for 50% smaller models:

• Reduce layer width: model still dense, requires no special hardware
• Block-sparsity: easier to accelerate
• Unstructured fine-grained sparsity: upper bound on accuracy

Let’s compare with 2:4 structured sparsity
BERT-LARGE CASE STUDY

Simpler Networks

Note: Validation loss is not final accuracy, but it can show general trends in network quality.
Halving the hidden size of encoders gives a smaller, dense network that is simple to accelerate, but the network itself is much worse.
Pruning the full network to 50% sparsity with 32x32 blocks then fine tuning can be accelerated on most parallel hardware, but the network performs poorly.

Note: For this and the following pruning techniques, we use the same model size - no growing the model as we prune.
Structured Sparsity is easy to accelerate with A100 and converges to nearly the same loss - final accuracy on SQuAD v1.1 is equivalent to dense.
BERT-LARGE CASE STUDY

Simpler Networks - Fine-Tuned

 Completely unstructured, fine-grained sparsity has similar loss compared to enforcing a 2:4 structure, but at only 50% sparse, it is incredibly hard to exploit.
BERT-LARGE CASE STUDY

Simpler Networks - Fine-Tuned

75% unstructured sparsity could be accelerated with standard techniques, but it is still tricky.

However, it does not approach the quality of the dense baseline.
Of these options, **2:4 structured sparsity** is the only technique that both maintains network quality and is easy to accelerate on A100.
ASP: AUTOMATIC SPARSITY FOR RETRAINING IN FRAMEWORKS
GENERATE A STRUCTURED SPARSE NETWORK

APEX’s Automatic SParsity: ASP

Conceptually simple - 3 step recipe
Simple in practice - 3 lines of code

NVIDIA’s APEX library

AMP = Automatic Mixed Precision

ASP = Automatic SParsity
import torch

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs)  # Define model structure

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # Define optimizer

x, y = DataLoader(...)  # load data samples and labels to train the model

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'dense_model.pth')
GENERATE A STRUCTURED SPARSE NETWORK

APEX’s Automatic SParsity: ASP

```python
import torch
from apex.contrib.sparsity import ASP

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs)  # Define model structure

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # Define optimizer

x, y = DataLoader(...)  # load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth')  # checkpoint has weights and masks
```
import torch
from apex.contrib.sparsity import ASP

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs)  # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # Define optimizer

x, y = DataLoader(...)  # Load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth')  # Checkpoint has weights and masks
import torch
from apex.contrib.sparsity import ASP

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs)  # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # Define optimizer

ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(...)  # load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth')  # checkpoint has weights and masks

PyTorch sparse fine-tuning loop

Generate a Structured Sparse Network
APEX's Automatic SParsity: ASP

Init mask buffers, tell optimizer to mask weights and gradients, compute sparse masks:
Universal Fine Tuning
import torch
from apex.contrib.sparsity import ASP

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth')) # Load pre-trained model

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

ASP.prune_trained_model(model)

x, y = DataLoader(...) # load data samples and labels to train the model

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
DIRECTIONS FOR FURTHER RESEARCH
SHORTEN RETRAINING

For some networks we were able to shorten retraining (Step-3) to a fraction of Step-1.

However, these shortened hyper-parameters didn’t apply to all networks.

**Further research:** investigate shorter, universal recipes.

<table>
<thead>
<tr>
<th>Network</th>
<th>Fine-Tuning Epochs</th>
<th>Accuracy</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>Reduced</td>
<td>Dense FP16</td>
<td>Sparse FP16</td>
<td>Short Sparse INT8</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>90</td>
<td>15</td>
<td>76.6</td>
<td>76.8</td>
<td>76.6</td>
</tr>
<tr>
<td>Inception v3</td>
<td>90</td>
<td>30</td>
<td>77.1</td>
<td>77.1</td>
<td>77.0</td>
</tr>
<tr>
<td>DenseNet-161</td>
<td>90</td>
<td>15</td>
<td>78.8</td>
<td>78.8</td>
<td>78.8</td>
</tr>
</tbody>
</table>
ACCELERATE TRAINING WITH SPARSITY

Sparse Tensor Cores can accelerate Step-3 (sparse retraining)

Can we eliminate Step-1?

- Recipe for training with sparsity from scratch (randomly initialized weights)

Research questions:

- How long to train densely (“dense warmup”)?
- Whether to periodically re-prune, if so: how frequently?
- How to use sparsity to accelerate weight gradient computation?
  - Input matrices are dense (activations and activation gradients), output is weight gradients (could be sparse)

Lots of active research, but still lacking a simple, general recipe
We moved fine-grained weight sparsity from research to production

Fine-grained structured sparsity is:

- 50% sparse, 2 out of 4 elements are zero
- Accurate with our 3-step universal fine-tuning recipe
  - Simple recipe: train dense, prune, re-train sparse
  - Across many tasks, networks, optimizers
- Fast with the NVIDIA Ampere Architecture’s Sparse Tensor Cores
  - Up to 1.85x in individual layers
  - Up to 1.5x in end-to-end networks
- S22082: Mixed-Precision Training of Neural Networks  5/20  2:45pm PDT
- S21929: Tensor Core Performance on NVIDIA GPUs: The Ultimate Guide  5/21  9:00am PDT
- S21819: Optimizing Applications for NVIDIA Ampere GPU Architecture  5/21  10:15am PDT