SparseProp: Efficient Sparse Backpropagation for Faster Training of Neural Networks at the Edge

Mahdi Nikdan [*1] **Tommaso Pegolotti** [*2] **Eugenia Iofinova** [1] **Eldar Kurtic** [1] **Dan Alistarh** [1 3]

Abstract

We provide an efficient implementation of the backpropagation algorithm, specialized to the case where the weights of the neural network being trained are *sparse*. Our algorithm is general, as it applies to arbitrary (unstructured) sparsity and common layer types (e.g., convolutional or linear). We provide a fast vectorized implementation on commodity CPUs, and show that it can yield speedups in end-to-end runtime experiments, both in transfer learning using already-sparsified networks, and in training sparse networks from scratch. Thus, our results provide the first support for sparse training on commodity hardware.

1. Introduction

The significant computational costs of deep learning have led to massive interest in approaches for leveraging *sparsity* in neural networks, which have been investigated in great breadth and depth (Hoefler et al., 2021). On the inference side, there is already emerging algorithmic and system support for sparsity on both GPUs (Mishra et al., 2021; Gale et al., 2020a) and CPUs (Elsen et al., 2020; NeuralMagic, 2022), as well as a wide range of methods for obtaining models which are both highly-sparse and highly-accurate.

A new frontier in the area is *accurate and efficient sparse training*. On the algorithmic side, there are several interesting proposals for *sparse training* algorithms (Dettmers & Zettlemoyer, 2019; Kusupati et al., 2020; Evci et al., 2020; Jayakumar et al., 2020; Schwarz et al., 2021), i.e. variants of stochastic gradient descent (SGD) which aim to keep as many weights as possible sparse during training. Another interesting related approach is *sparse transfer* (Zafrir et al., 2021; Chen et al., 2021; Iofinova et al., 2022; Kurtic et al., 2022), by which models sparsified on a large pretraining corpus are then used for *transfer learning* on different tasks, while preserving the sparsity mask.

Despite this progress on the optimization side, the vast majority of these approaches lack *system support for fast training*, in that they do not provide any practical speedups. This is because the weight sparsity introduced is *unstructured*, which is notoriously hard to leverage for computational gains. Specifically, there is no general implementation of backpropagation that can leverage unstructured weight sparsity for practical speedup on common hardware. At the same time, approaches leveraging *block sparsity* (Mishra et al., 2021; Gray et al., 2017) can only reach lower sparsity without significant accuracy drops, and require specialized training algorithms (Lagunas et al., 2021; Jiang et al., 2022). As such, unstructured weight sparsity is often dismissed as a practical way of accelerating model training.

Contribution. We contradict this conventional wisdom by presenting a new vectorized implementation of backpropagation (Rumelhart et al., 1986), designed to be efficient in the case where the weights of the neural network are *sparse*, i.e. contain a significant fraction of zero values, and show its potential for practical speedups in common edge training scenarios, for both vision and language tasks. Our algorithm, called SparseProp, is general in the sense that 1) it applies to arbitrary sparsity patterns, 2) general layer types, and 3) can be efficiently vectorized using standard CPU-supported approaches. The asymptotic complexity of the algorithm is *linear* in the *layer density*, i.e. the number of non-zero weights in the layer, providing proportional runtime improvements to the weight sparsity, for both linear and convolutional layers.

To illustrate practical efficiency, we provide a fast vectorized implementation of SparseProp aimed at general-purpose Intel and AMD CPU architectures. Specifically, our implementation provides drop-in replacement implementations for standard layer types, and only relies on widely-supported AVX2 instructions. We show that SparseProp can lead to practical runtime improvements both on single sparse layers, validating our linear sparsity scaling claims, as well as on end-to-end training of sparse models.

We provide results for an integration with Pytorch (Paszke

[*]Equal contribution [1]IST Austria [2]ETH Zurich [3]Neural Magic, Inc.. Correspondence to: Mahdi Nikdan <mahdi.nikdan@ist.ac.at>, Tommaso Pegolotti <tommaso.pegolotti@inf.ethz.ch>, Dan Alistarh <dan.alistarh@ist.ac.at>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

et al., 2019a), which can run sparse backpropagation for linear and convolutional layers, covering most popular model families. As such, SparseProp can provide direct support for pruning-during training methods like Gradual Pruning (Zhu & Gupta, 2017), RigL (Evci et al., 2020) or AC/DC (Peste et al., 2021), which assume a fixed sparsity mask for any forward / backward pass, and can be modified to support more complex methods (Jayakumar et al., 2020), which specify different sparsities for weights and gradients.

Our end-to-end experiments aim to make the case that sparsity can be a viable option for DNN training *at the edge*. That is, we explore settings where a device with moderate computational power (e.g., a CPU with a limited number of cores) performs either *sparse transfer* or *from-scratch sparse training* over a specialized task. We investigate sparsity-versus-accuracy trade-offs in two model/task combinations: 1) ResNets (He et al., 2016) applied to twelve popular vision tasks (Kornblith et al., 2019), and 2) a standard BERT-base model (Devlin et al., 2019) applied to GLUE language modelling tasks (Wang et al., 2018).

In the *sparse transfer* scenario, we are provided an already-sparse model pretrained on a large corpus, e.g. ImageNet (Russakovsky et al., 2015) respectively WikiText (Merity et al., 2016), and wish to finetune the corresponding sparse weights on a (usually smaller) target dataset. This application has gained significant popularity (Zafrir et al., 2021; Chen et al., 2021; Iofinova et al., 2022; Kurtic et al., 2022), and pretrained sparse models are available for several standard tasks and models (Wolf et al., 2019; SparseZoo, 2022). In this context, we show that, for both vision and language tasks, SparseProp can lead to end-to-end sparse transfer speedups of up to 1.85x, at similar accuracies, relative to CPU-based finetuning of *dense models* in the same environment. Measured only over backward-pass operations—and thus omitting framework-level overheads–our algorithm provides speedups of 3.6x at 95% model sparsity.

In the second scenario, we examine the ability of SparseProp to provide speedups for *sparse training from scratch*, on the same series of tasks, adapting variants of sparse training (Zhu & Gupta, 2017) to our setting. Experiments show that, in this scenario, SparseProp leads to end-to-end speedups of up to 1.4x, with moderate accuracy loss.

In sum, our results show that SparseProp can efficiently provide system support for CPU-based unstructured sparse training, ensuring speedups for both from-scratch training and sparse transfer. We believe our approach could lead to additional practical impact for research on sparsity, especially given that our end-to-end runtime numbers can still be improved via additional optimizations, and by mitigating external, framework-specific overheads. Our implementation and examples are publicly available at our github page: https://github.com/IST-DASLab/sparseprop.

2. Related Work

Sparse Inference. One of the key motivations behind sparsity in DNNs is reducing inference costs. For this, an impressive number of weight pruning techniques have been introduced, e.g. (LeCun et al., 1990; Hagiwara, 1994; Han et al., 2016b; Singh & Alistarh, 2020; Sanh et al., 2020). Complementing this work, there have been a number of algorithmic proposals for efficient sparse inference algorithms over DNNs, e.g. (Park et al., 2016; Han et al., 2016a; Gale et al., 2020a; Elsen et al., 2020), although it is known that layer-wise gains can be difficult to translate into end-to-end speedups (Wang, 2020). Nevertheless, sparse inference support is now available on both CPUs, e.g. (NeuralMagic, 2022) and GPUs (Mishra et al., 2021).

Sparse SGD-Based Training. As noted, there has been a significant amount of work on SGD-like algorithms for *sparse training* of DNNs, balancing accuracy while trying to maximize sparsity in the models' internal representations (Zhu & Gupta, 2017; Lis et al., 2019; Dettmers & Zettlemoyer, 2019; Zhang et al., 2020; Wiedemann et al., 2020; Kusupati et al., 2020; Evci et al., 2020; Jayakumar et al., 2020; Peste et al., 2021; Schwarz et al., 2021). Unfortunately, a precise comparison is quite difficult, since each makes different assumptions regarding the degree of sparsity in the network's internal representations, potentially even varying the amount of sparsity between weights and gradients (Jayakumar et al., 2020).

Hubara et al. (2021) proposed a theoretically-justified approach for identifying sparse transposable masks matching the NVIDIA 2:4 sparsity pattern, which could be leveraged for faster training on GPUs. We do note however that, currently, GPU-based 2:4 sparsity speedups tend to be minimal (NVIDIA, 2021).

Sparse Training for Speedup. Leveraging sparsity for practical speedups has been a major goal in model compression (Hoefler et al., 2021). Yang et al. (2020) proposed a specialized hardware accelerator which is specialized to the DropBack pruning algorithm (Lis et al., 2019). SWAT (Raihan & Aamodt, 2020) proposed a sparsity-aware algorithm for the case where both weights and activations have high sparsity, and showed speedups in a simulated environment. Their approach works only for specific networks, and can lose significant accuracy. Zhou et al. (2021) introduced a variance reduced gradient policy estimator for updating the structure of the sparse network efficiently during training, but only focus on structured sparsity case.

More recently, Jiang et al. (2022) proposed an algorithm-hardware co-design approach, for the case of GPU-based training. Specifically, their approach imposes *block sparsity* in GPU-friendly patterns, and leverages it for speedup.

By contrast to this work, our approach considers efficient support for backpropagation for *unstructured sparse*

vload(address)	load from memory address
vstore(address, a)	store a at memory address
vbroadcast(a)	fill a register with a
vfmadd(a, b, c)	return $a \cdot b + c$
vaddreduce(a)	return sum elements of a

Table 1. List of vector instructions used in the implementation and their semantics.

weights, implements this efficiently for commodity CPUs, and shows that this can be leveraged for *end-to-end speedup* during training. Specifically, this provides support to the vast amount of existing work on unstructured sparse training algorithms, on commodity hardware.

System Support. Pytorch (Paszke et al., 2019b) introduced partial sparse tensor support, while the STen (Ivanov et al., 2022) provides a general interface for such representations. Our work is complementary, as our implementation can be interfaced with Pytorch for speedups.

3. The Sparse Backpropagation Algorithm

3.1. Background

SIMD Instructions. Fast and efficient numerical code heavily relies on Single Instruction Multiple Data (SIMD) instructions to improve performance. These instructions operate on specialized machine registers (xmm, ymm, and zmm) that contain multiple values.

Our implementations currently support x86 machines that provide the standard AVX2 instruction set, which uses 256 bit registers, or 8 single precision floating point values. Table 1 provides an overview of the instructions employed by our library. Our SIMD implementation structure follows the Load-Compute-Store paradigm, where data is explicitly transferred to registers via the **loadv** and **broadcastv** instructions. Computation is performed on the data in the registers using fused multiply-add instructions **vfmadd** ($r = a \cdot b + c$), and the results are subsequently moved back to memory with the **vstore** instruction. Following this structure, we significantly increase performance since the data loaded into the registers can be used for multiple operations before being stored back in memory.

Backpropagation. Let $f(\mathbf{X}; \mathbf{W})$ represent a layer (fully-connected or convolution) in a neural network $\mathcal{N}$; $\mathbf{W}$ represents the parameters of this layer and $\mathbf{X}$ represents a batch of inputs. Let B be the batch size. Additionally, denote the output of this layer by $\mathbf{O} = f(\mathbf{X}; \mathbf{W})$. Let L be the loss of the whole network $\mathcal{N}$ for this batch of inputs. Backpropagating through this layer involves calculating the gradients $\partial L/\partial \mathbf{W}$ and $\partial L/\partial \mathbf{X}$, given $\partial L/\partial \mathbf{O}$.

Consider the situation where we have a highly sparsified matrix $\mathbf{W}$ that is stored as a sparse matrix. During the backpropagation process, it is necessary to calculate the gradient of this matrix. However, in practice, the full gradient of the dense matrix is often calculated, even though the pruned elements are not updated and their gradients are discarded. This can be inefficient, as it consumes a significant amount of computation and time without providing any benefits.

3.2. The Case of Fully-connected Layers

We now focus on the case where $f(.)$ is a fully-connected layer. Assume $\mathbf{X}$ and $\mathbf{W}$ are $B \times M$ and $M \times N$ matrices, respectively. Consequently, $\mathbf{O} = f(\mathbf{X}; \mathbf{W}) = \mathbf{X}\mathbf{W}$ will be a $B \times N$ matrix, ignoring the bias term for simplicity. The gradients of L with respect to $\mathbf{X}$ and $\mathbf{W}$ are calculated as:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{O}} \mathbf{W}^T \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial L}{\partial \mathbf{O}} \tag{2}$$

If we examine equations (1) and (2), we can see that the former is a General Sparse Matrix-Matrix Multiplication (SpGEMM) operation, while the latter is a Sampled Dense Dense Matrix Multiplication (SDDMM) operation.

Sparse Representation. The matrix $\mathbf{W}$ is stored in a compressed sparse row (CSR) format, a standard representation for sparse matrices. The non-zero values of the matrix are stored in the arrays W_{vals} and W_{cols}, which correspond to the values and column indices of the non-zero elements, respectively. The array W_{rows} encodes each row's start and end indices. For example, the non-zero values and column indices of a row i of $\mathbf{W}$ are contained between positions $W_{\text{rows}}[i]$ and $W_{\text{rows}}[i+1]$.

Algorithm. In Algorithm 1, we present high-level pseudocode for backpropagation in our linear layer. The calculations for (1) and (2) are performed in a single pass by utilizing the sparsity pattern of $\mathbf{W}$, which is identical to $\partial L/\partial \mathbf{W}$. Specifically, the result of $(\partial L/\partial \mathbf{O})\mathbf{W}^T$ is computed as a sparse matrix-matrix multiplication. Whereas $\mathbf{X}^T(\partial L/\partial \mathbf{O})$ is computed as an SDDMM, with nnz dot-products, where nnz is the number of non-zero elements of $\mathbf{W}$. In more detail, the computation is divided into 3 loops. The innermost loop contains the core of the computation. It computes at each iteration 16 floating point operations using 2 **fmadd** instructions: the first **fmadd** computes 8 entries of $\partial L/\partial \mathbf{X}$ and the second accumulate a dot-product in a register acc.

Implementation Details. We operate on the transposed matrices in our linear layer implementation to improve cache utilization. Specifically, in both the forward and backward passes, we operate on the transposed version of the input matrix, $\mathbf{X}^T$, which is a column-major representation of $\mathbf{X}$.

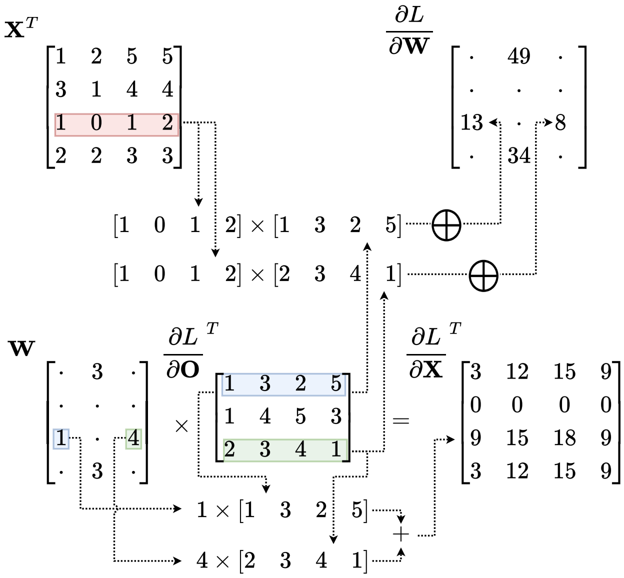


Figure 1. Visual representation of the core computation of Algorithm 1 using vector registers of size 4. We represent elementwise multiplication with $\times$ and **vaddreduce** with $\bigoplus$.

By doing so, we achieve a streaming access pattern over the rows of $\mathbf{X}^T$, as we see from the innermost loop Algorithm 1. Additionally, we leverage that the transpose of a CSR matrix is none other than the same matrix in Compressed-Sparse-Column (CSC) format to avoid expensive sparse transpose operations. An example computation is given in Figure 1.

3.3. Sparse Backpropagation for Convolutional Layers

We now examine convolutional layers. Denote the number of input channels by IC, and the number of output channels by OC. The input width and height are represented by M and N, respectively, and let the kernel size be $K \times K$. The input tensor, $\mathbf{X}$, and the weights tensor, $\mathbf{W}$, have dimensions $B \times IC \times M \times N$ and $OC \times IC \times K \times K$, respectively. For simplicity, here we only consider the case where padding is 0 and stride is 1. For larger padding and stride, the generalization is not difficult.

The output tensor, $\mathbf{O}$, will be of size $B \times OC \times OM \times ON$, where $OM = M - K + 1$ and $ON = N - K + 1$. For $0 \leq b < B, 0 \leq oc < OC, 0 \leq p < OM, 0 \leq q < ON$ we have:

$$\mathbf{O}[b, oc, p, q] = \sum_{ic=0}^{IC-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \mathbf{W}[oc, ic, i, j] \cdot \mathbf{X}[b, ic, p+i, q+j]. \tag{3}$$

Using the chain rule, it is easy to check that for $0 \leq b < B, 0 \leq ic < IC, 0 \leq m < M, 0 \leq n < N$:

$$\frac{\partial L}{\partial \mathbf{X}}[b, ic, m, n] = \sum_{oc=0}^{OC-1} \sum_{p=p_s}^{m} \sum_{q=q_s}^{n} \frac{\partial L}{\partial \mathbf{O}}[b, oc, p, q] \cdot \mathbf{W}[oc, ic, m-p, n-q], \tag{4}$$

with $p_s = m - K + 1$, and $q_s = n - K + 1$. And for $0 \leq oc < OC, 0 \leq ic < IC, 0 \leq i < K, 0 \leq j < K$, we have:

$$\frac{\partial L}{\partial \mathbf{W}}[oc, ic, i, j] = \sum_{b=0}^{B-1} \sum_{p=0}^{M-K} \sum_{q=0}^{N-K} \frac{\partial L}{\partial \mathbf{O}}[b, oc, p, q] \cdot \mathbf{X}[b, ic, p+i, q+j]. \tag{5}$$

It is assumed that the weight tensor $\mathbf{W}$ is sparse. In accordance with equation (4), when a weight $\mathbf{W}[oc, ic, m-p, n-q]$ is pruned, the multiplication and corresponding addition operations can be skipped. Furthermore, when a weight $\mathbf{W}[oc, ic, i, j]$ is pruned, the calculation of the gradient for this parameter is not necessary, as it will not be updated, and therefore the computation outlined in equation (5) can be skipped.

Sparse Representation. To efficiently represent a sparse tensor, we employ a representation akin to the compressed sparse row (CSR) format used for sparse matrices. Four arrays, W_{och}, W_{ich}, W_{x}, and W_{y}, are used to store the indices, and an array W_{vals} is used to store the non-zero values. Specifically, W_{x} and W_{y} are arrays of size nnz, which contain the coordinates of the non-zero values of each filter. W_{och} is an array of size $OC + 1$, which encodes the start of each output channel's entries in W_{ich}. Finally, W_{ich}

Algorithm 1 AVX2 Linear Backward Pass

for $i = 0$ **to** $N - 1$ **do**
  for $j = \mathbf{W}_{\text{cols}}[i]$ **to** $\mathbf{W}_{\text{cols}}[i+1]$ **do**
    /* `repeat one weight entry 8 times` */
    $v \leftarrow$ **vbroadcast**$(\mathbf{W}_{\text{vals}}[j])$
    /* `initialize` acc `to zero` */
    $acc \leftarrow$ **vbroadcast**(0)
    $r \leftarrow \mathbf{W}_{\text{rows}}[j]$
    for $k = 0$ **to** $B - 1$ **with** $k \mathrel{+}= 8$ **do**
      /* `load 8 values` */
      $dx \leftarrow$ **vload**$((\partial L/\partial \mathbf{X})_{r,k})$
      $do \leftarrow$ **vload**$((\partial L/\partial \mathbf{O})_{r,k})$
      $x \leftarrow$ **vload**$(\mathbf{X}_{i,k})$
      /* `compute 8` $dx = do \cdot v + dx$ */
      $dx \leftarrow$ **vfmadd**(do, v, dx)
      /* `compute 8` $acc = do \cdot x + acc$ */
      $acc \leftarrow$ **vfmadd**(do, x, acc)
      /* `store updated` dx `back` */
      vstore$((\partial L/\partial \mathbf{X})_{r,k}, dx)$
    end for
    /* `sum the 8 values in` acc */
    $(\partial L/\partial \mathbf{W})_{\text{vals}}[j] \leftarrow$ **vaddreduce**(acc)
  end for
end for

is an array of size $OC \times (IC+1)$, which encodes the indices in W_{x}, W_{y}, and W_{vals} of each input channel. For example, for an output channel oc, the non-zero elements of the input channel ic are stored between indices $W_{\mathrm{och}}[oc] + W_{\mathrm{ich}}[oc \cdot (IC+1)+ic]$ and $W_{\mathrm{och}}[oc]+W_{\mathrm{ich}}[oc\cdot(IC+1)+ic+1]$. As example, consider the following sparse tensor of dimensions $(3, 2, 2, 3)$

$$\left\{\begin{matrix} \begin{bmatrix} \cdot & a & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} & \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} & \begin{bmatrix} \cdot & \cdot & \cdot \\ b & \cdot & c \end{bmatrix} \\ \\ \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} & \begin{bmatrix} \cdot & d & \cdot \\ \cdot & e & \cdot \end{bmatrix} & \begin{bmatrix} \cdot & \cdot & f \\ \cdot & \cdot & \cdot \end{bmatrix} \end{matrix}\right\},$$

where $\cdot$ represents a zero value. Its sparse representation is given by

$$\begin{aligned} W_{\mathrm{och}} &= \begin{pmatrix} 0 & 1 & 3 & 6 \end{pmatrix}, \\ W_{\mathrm{ich}} &= \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 2 & 0 & 2 & 3 \end{pmatrix}, \\ W_{\mathrm{x}} &= \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}, \\ W_{\mathrm{y}} &= \begin{pmatrix} 1 & 1 & 1 & 0 & 2 & 2 \end{pmatrix}, \\ W_{\mathrm{vals}} &= \begin{pmatrix} a & b & c & d & e & f \end{pmatrix}. \end{aligned}$$

Although the usage of a W_{och} may seem superfluous, its inclusion allows us to reduce total memory usage. Indeed, assuming $K < 256$ and $IC < 8192$, we can store entries in W_{x} and W_{y} using `uint8_t` and W_{ich} using `int16_t`. Therefore, using W_{och} lowers memory usage from $4B \times (IC + 1) \times OC$ bytes to $2B \times (IC + 1) \times OC + 4 \times OC$ bytes.

Algorithm. In Algorithm 2, we present an overview of our backpropagation algorithm for a convolutional layer. If we ignore at first the pointer arithmetic needed to traverse the structures, the main structure remains similar to that of Algorithm 1 as both innermost loops use the same instructions.

Implementation Details. We developed two kernels for fast 2D convolutions based on the input dimensions. For larger M and N, we found that no preprocessing was needed. Keeping the input dimensions as $B \times IC \times M \times N$ offers both computing batches in parallel on multiple threads and high single-core parallelization using AVX2 instructions. On the other hand, for small M and N, which often occur for the last layers of a network, we found it more efficient to permute the tensors to have the batch as the last index. In particular, $\mathbf{X}$ and $\partial L/\partial\mathbf{X}$ became of size $IC \times M \times N \times B$, and $\mathbf{O}$ and $\partial L/\partial\mathbf{O}$ became of size $OC \times OM \times ON \times B$. Indeed, setting B as the last dimension allows the usage of SIMD instructions even for small M and N.

Algorithm 2 AVX2 Convolutional Backward Pass

```
for ic = 0 to IC do
  for oc = 0 to OC do
    si_s = W_och[oc] + W_ich[ic]
    si_e = W_och[oc] + W_ich[ic + 1]
    for si = si_s to si_e do
      /* repeat 8 times */
      v ← vbroadcast(W_vals[j])
      /* initialize acc to zero */
      acc ← vbroadcast(0)
      p_s ← max(0, pad − W_x[si])
      p_e ← min(pad − W_x[si] + M, OM)
      q_s ← max(0, pad − W_y[si])
      q_e ← min(pad − W_y[si] + N, ON)
      for p = p_s to p_e do
        for q = q_s to q_e do
          for k = 0 to B − 1 with k += 8 do
            /* load 8 values */
            do ← vload((∂L/∂O)_{oc,p,q,k})
            dx ← vload((∂L/∂X)_{ic,p,q,k})
            x ← vload(X_{ic,p,q,k})
            /* compute 8 dx = do · v + dx */
            dx ← vfmadd(do, v, dx)
            /* compute 8 acc = do · x + acc */
            acc ← vfmadd(do, x, acc)
            /* store the updated dx */
            vstore((∂L/∂X)_{ic,p,q,k}, dx)
          end for
        end for
      end for
    end for
    /* sum the 8 values in acc */
    (∂L/∂W)_vals(j) ← vaddreduce(acc)
  end for
end for
```

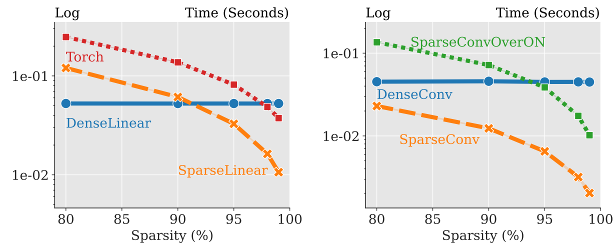
4. Experiments

Setup and Goals. We now experimentally validate our approach. First, we perform an in-depth exploration of our algorithm's runtime relative to weight sparsity in a synthetic scenario, i.e. for standard layer and input shapes. Then, we examine performance for two *end-to-end training* scenarios, as part of a Pytorch integration. Specifically, we examine performance for *sparse transfer*, i.e. fine-tuning of already-sparsified accurate models on a different "transfer" dataset, and *from-scratch sparse training* on some specialized tasks.

Pytorch Modules. We provide two Pytorch modules, one for linear and one for convolution layers, facilitating the integration of our algorithm to different applications. To this end, we employ the pybind11 library (Jakob et al., 2016) for fast communication between Pytorch and the C++ backend.

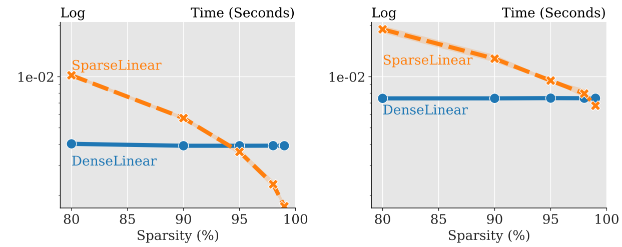
4.1. Synthetic Performance Evaluation

This section analyzes the runtime scaling for our linear and convolutional layers implementations of Algorithms 1 and 2. To validate our linear-scaling claims, we specifically examine the runtime dependence of our implementations relative to sparsity. These implementations are compared to their dense counterparts available in Pytorch. Additionally, for the linear layer implementation, we compare it against the sparse implementation offered by Pytorch. We present



(a) Linear backward (b) Convolutional backward

Figure 2. Runtime of our single-core backward propagation of sparse linear and convolutional layers. The former has dimension $(M, N) = (768, 3072)$ and the input is of size $(B, M) = (902, 768)$, while the latter is of size $(OC, IC, K, K) = (256, 128, 3, 3)$, and the input is $(B, IC, M, N) = (8, 128, 7, 7)$.



(a) Parallel linear forward (b) Parallel linear backward

Figure 3. Runtime measurement of our parallel sparse linear layer compared against a dense implementation. The layer has dimension $(M, N) = (768, 3072)$ the input is of size $(B, M) = (902, 768)$.

the improvement in our implementations' performance as a function of a layer's sparsity ranging from 80% to 99%. In particular, we give the runtime for our multithreaded implementations run on 8-threads in Figures 4 and 3, and the runtime for the single-core implementations in Figure 2. To highlight the speedup between implementations, we give all our measurements in *log-lin* plots.

Linear layers. We evaluate the performance of our linear layer by measuring the runtime of forward and backward pass through a large layer of dimensions $(M, N) = (768, 3072)$ and an input of size $(B, M) = (902, 768)$. We report the single-core results for the backward pass in Figure 2(a), and we compare them to a Pytorch-based sparse implementation, which is currently only single-core on CPUs. Our sequential performance increases linearly with the sparsity of $\mathbf{W}$, we match a dense implementation around 90% sparsity, and we obtain a $5\times$ speedup at 99%. In Figure 3, we plot the results for the multithreaded forward and backward passes. For our multithreaded forward pass, we observe similar behavior to the single-core variant beating the dense implementation after 95% sparsity. It should be noted that the performance of our multithreaded implementation of the backward pass is currently limited by synchronization overheads, which can be removed with additional optimizations. As of now, it matches the dense implementation only at very high levels of sparsity.

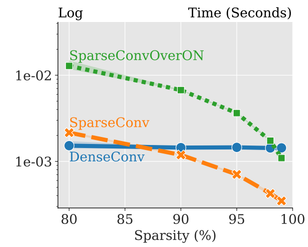
Convolutional layers. In the case of the convolutional layers, we present the runtime performance for two distinct scenarios: small input tensor and large input tensor. The dimensions of the layer are set to $(OC, IC, K, K) = (256, 128, 3, 3)$, and the input tensor's dimensions are set to $(B, IC, M, N) = (8, 128, 7, 7)$ and $(B, IC, M, N) = (32, 256, 244, 244)$. These scenarios highlight the difference between the two sparse convolution kernels we developed: `SparseConv` and `SparseConvOverON`. The former is presented in Algorithm 2, and the latter is its permuted variant that vectorizes over the ON dimension of the tensor.

We show that by permuting the input, we substantially improve the performance of our algorithm. The single-core runtime performance over a small input is presented in Figure 2(b), where we observe a significant speedup compared to a dense implementation, with 19x speedup at 99% sparsity. The parallel runtime for forward and backward passes are in Figure 4. Timings for small input sizes are given in Figures 4(a) and 4(b), and for large inputs in Figures 4(c) and 4(d). We see how permuting impacts our performance for small ON values achieving a speedup over the dense implementation of up to 5x for the forward and backward pass at 99% sparsity. On the other hand, for larger M and N, our results indicate that vectorizing over ON yields the best performance, resulting in a speedup of 3.35x for the forward pass and 9x for the backward.

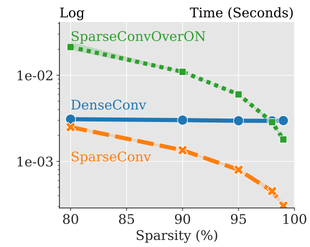
4.2. End-to-End Training Experiments

We now evaluate SparseProp on *sparse transfer learning*, and *sparse training from scratch*. In each case we examine sparsity settings which, while maintaining reasonable accuracy, can achieve non-trivial speedups. Since we executed over more than 15 different tasks, on multiple models, the full experiments used to determine accuracy are executed on GPU. At the same time, we computed CPU speedups using proportionally-shortened versions of the training schedules on CPU, and validated correctness in a few end-to-end runs on different tasks.

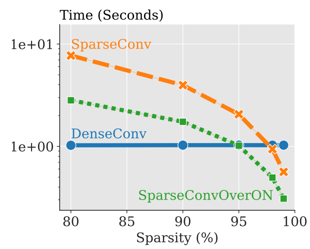
In all the experiments, dense modules (linear or convolution) are executed dense as long as they are less than 80% sparse. Once a module reaches at least 80% sparsity (which may happen during training for gradual pruning scenarios), we measure the time to run one batch through both dense and sparse versions, and choose the fastest version based on forward+backward time (in the case of convolution, both sparse implementations are considered). Notice that in all experiments, the sparsity patterns change only a few times during the whole run, meaning the overhead of these few extra batches is negligible. (An alternative approach would



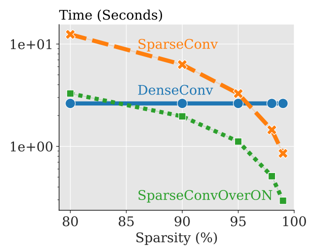
(a) Parallel Conv. forward



(b) Parallel Conv. backward



(c) Parallel Conv. forward



(d) Parallel Conv. backward

Figure 4. Runtime measurement of our parallel sparse convolutional layers (base and vectorized over the ON dimension) on 8 threads compared against a dense implementation. The layer has dimension $(OC, IC, K, K) = (256, 128, 3, 3)$, and the input's dimensions are set to $(B, IC, M, N) = (8, 128, 7, 7)$ in (a) and (b) and to $(B, IC, M, N) = (32, 256, 244, 244)$ in (c) and (d).

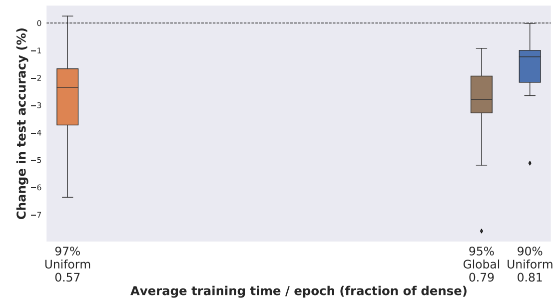


Figure 5. Accuracy vs. speedup for *transfer learning* experiments on the ResNet50 architecture. The boxplots represent aggregated performance across all twelve target tasks.

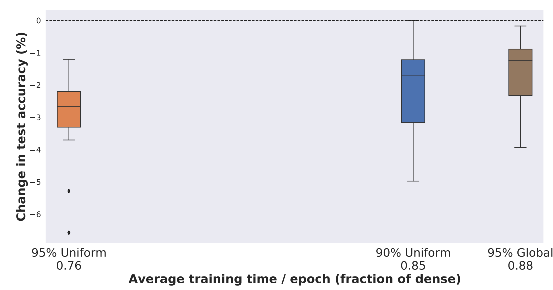


Figure 6. Accuracy vs. estimated speedup for *from-scratch learning* experiments on the ResNet18 architecture. The boxplots represent aggregated performance across all twelve target tasks.

be to generate a static database of the best implementation choices for each layer type and size, and greedily adopt the implementation for each layer in turn.)

4.2.1. Application 1: Sparse Transfer

Image Classification. We first consider a standard transfer learning setup for image classification using CNNs, in which a model pretrained on ImageNet-1K has its last (FC) layer resized and re-initialized, and then is further finetuned on a smaller target dataset. As targets, we focus on twelve datasets that are conventionally used as benchmarks for transfer learning, e.g. in (Kornblith et al., 2019; Salman et al., 2020; Iofinova et al., 2022). See Table 7 for a summary of the tasks. Importantly, input images are scaled to standard ImageNet size, i.e. $224 \times 224 \times 3$, resulting in proportional computational costs.

We consider dense and sparse ResNet50 models pre-trained (and sparsified) on the ImageNet-1K dataset. Models are pruned using the AC/DC method (Peste et al., 2021), which shows high accuracy on ImageNet-1K, and produces transferrable sparse features (Iofinova et al., 2022). (We adopt their publicly-available models.) To explore the accuracy-vs-speedup trade-off, we consider both a *Uniform pruning* scenario, in which all convolutional layers except for the input are pruned to a uniform sparsity (90% and 97%), and a *Global pruning* scenario, in which all convolutional layers are pruned jointly, to a target average sparsity (95%), using the global magnitude pruning criterion. The former two models have been trained for 200 epochs each, and have 76.01% and 74.12% Top-1 accuracy, whereas the latter is trained for 100 epochs and has 73.1% Top-1 accuracy. The dense model has 76.8% Top-1 accuracy. For transfer learning, we maintain the sparsity pattern of the models, reinitialize the final fully-connected layer, and train for 150 epochs on each of the 12 "downstream" tasks.

In Figure 5, we aggregated results across all tasks, in terms of mean and variance of the *accuracy drop relative to transferring the dense model* (the full per-task accuracy results are presented in Table 8, and speedups are presented in Table 2). As expected, the aggregated sparse test accuracy drops relative to the dense baseline, proportionally to the ImageNet Top-1 accuracy. The Uniform-90 model shows the smallest drops (1% on average), but also the lowest end-to-end speedup (25%), while the Uniform-97 and Global-95 models have slightly worse average drops (around 2%). Remarkably, due to higher initial accuracy, the Uniform-97 model has similar accuracy to Global-95, but much higher end-to-end speedup of 1.75x.

Case Study: 95% Uniformly-Pruned ResNet18. We now analyze in detail both the accuracy drops and the per-layer and global speedups for a 95% uniformly-pruned ResNet18

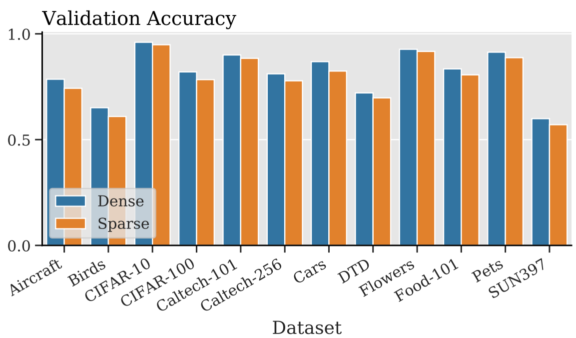


Figure 7. Top-1 validation transfer accuracy of dense and 95% sparse ResNet18 models pre-trained on ImageNet-1K.

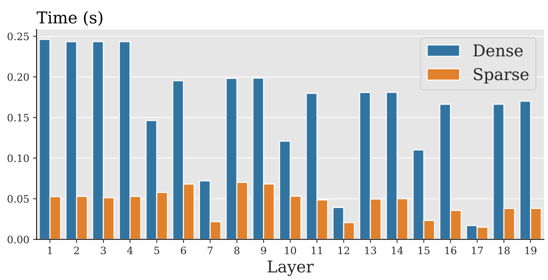


Figure 8. Layer-wise back-propagation time comparison between Dense and 95% uniformly-pruned Resnet18. Note that first and last layers are always dense and are hence removed from the comparison.

model. On ImageNet, the AC/DC pruned model ResNet18 model has 68.9% Top-1 accuracy, a relative 1% drop from the Torchvision baseline. Figure 7 depicts the transfer performance of the respective models on twelve target datasets, using exactly the same transfer recipe for both sparse and dense models. The accuracy loss is moderate (2.85% average, with a maximum gap of 4.5% Top-1, across all tasks).

Figure 8 depicts layer-wise backward speedups with respect to Pytorch's dense implementation. The results show that the overall end-to-end speedup is 1.58x, with a 1.26x speedup end-to-end for forward computations and a 2.11x speedup end-to-end for backward computations. A closer examination reveals that if we only measure the time spent in convolution and linear modules' forward and backward functions we get a 2.53x speedup, suggesting the presence of significant overheads outside of the convolution and linear computations (such as batch normalization layers and ReLU) for the Pytorch CPU implementation. More precisely, our implementations provide speedups of 1.57x and 3.60x, for the forward and backward multiplications, respectively. This highlights the efficiency of our backward algorithms, but also the overheads of Pytorch's current CPU training implementation. (Specifically, in our sparse implementation, the batch normalization and ReLU computations,

ResNet50, batch size=64	Forward	Backward	End-to-End	Forward	Backward	End-to-End
Dense	10.98 ± 0.14	19.85 ± 0.10	$\mathbf{31.19 \pm 0.24}$	1.44 ± 0.09	3.41 ± 0.04	$\mathbf{5.09 \pm 0.09}$
Uniform 90%	1.11×	1.31×	**1.23**×	1.00×	1.26×	**1.16**×
Global 95%	1.24×	1.28×	**1.27**×	1.13×	1.18×	**1.15**×
Uniform 97%	1.59×	1.88×	**1.76**×	1.40×	1.76×	**1.59**×
Global 98%	1.42×	1.71×	**1.58**×	1.29×	1.61×	**1.46**×

Table 2. Single-core (left) and parallel (right) relative speedup for sparse ResNet50 models pretrained on ImageNet1k over a dense implementation. Times are reported in seconds.

BERT-base	Forward	Backward	End-to-End	MNLI	QQP
Dense	0.72 ± 0.01	1.33 ± 0.00	$\mathbf{2.67 \pm 0.01}$	84.54%	91.06%
Global 97%	1.27×	1.40×	**1.37**×	80.91%	90.33%

Table 3. Accuracies and single-core relative speedup for transfer learning sparse BERT-base models. Times are reported in seconds.

executed via Pytorch, take approximately 25% of the total training time.)

Language Modelling. Next, we replicate the setup of Kurtic et al. (2022), where a BERT-base (Devlin et al., 2019) model is pruned in the pre-training stage on BookCorpus and English Wikipedia (Lhoest et al., 2021) with the state-of-the-art unstructured pruner oBERT (Kurtic et al., 2022). After that, the remaining weights are fine-tuned on several downstream tasks with fixed sparsity masks. We consider a model with 97% global sparsity, and maintain its masks throughout finetuning on the MNLI and QQP tasks from the GLUE benchmark (Wang et al., 2018). Both accuracy and speedup results are shown in Table 3, and show 37% speedup on a single core for inference on this model, at the price of $\sim$ 1–3.5% accuracy drop.

4.2.2. Application 2: Sparse Training from Scratch

Image Classification. We evaluate SparseProp in the from-scratch sparse training scenario. We first consider the same 12 specialized datasets as in Section 4.2.1, and train a ResNet18 architecture *from scratch*. We apply 90% and 95% sparsity using Gradual Magnitude Pruning (Zhu & Gupta, 2017), in two scenarios—Uniform sparsity, in which all convolutional layers, except the first, are pruned to the same target, and the Global scenario, in which the magnitudes of the weights are aggregated across all layers, and the smallest-magnitude weights are pruned to reach the target sparsity. (For the latter scenario, we consider only 95% sparse models.) Note that in the Uniform scenario, we do not prune the initial convolution, nor the final FC layer,

	Forward	Backward	End-to-End	MNLI	QQP
Dense	0.72 ± 0.01	1.33 ± 0.00	$\mathbf{2.67 \pm 0.01}$	84.54%	91.06%
Uniform 95%	1.27×	1.40×	**1.37**×	80.91%	90.33%

Table 4. Accuracies and single-core relative speedup for transfer learning sparse BERT-base models. Times are reported in seconds.

	ResNet18, batch size=256				BERT-base, batch size=4			
	Forward	Backward	End-to-End	CelebA AUC	Forward	Backward	End-to-End	QQP
Dense	1×	1×	1×	80.2	1×	1×	1×	91.06
Global 90%	1.02×	1.14×	**1.08**×	81.6	-	-	-	-
Uniform 90%	0.99×	1.36×	**1.17**×	-	0.97×	0.96×	**1.04**×	90.09
Global 95%	1.04×	1.23×	**1.13**×	81.7	-	-	-	-
Uniform 95%	1.09×	1.57×	**1.32**×	-	1.16×	1.19×	**1.23**×	89.15
Uniform 97%	1.19×	1.65×	**1.41**×	81.1	1.24×	1.31×	**1.31**×	87.81
Global 99%	1.08×	1.48×	**1.28**×	81.0	-	-	-	-

Table 5. Relative speedup on the from-scratch training for sparse ResNet18 and BERT-base models over a dense implementation.

ResNet18, batch size=256	Forward	Backward	End-to-End	Cifar-10 Accuracy
Dense	1×	1×	1×	94.3%
Uniform 90%	0.95×	1.40×	**1.20**×	92.9%
Uniform 95%	1.19×	1.85×	**1.53**×	92.2%
Uniform 98%	1.50×	2.32×	**1.92**×	90.7%

Table 6. Relative speedup on the from-scratch RigL training for sparse ResNet18 models over a dense implementation.

while in the Global scenario, we do. In both cases, we train the model dense for ten epochs before pruning the lowest 5% of the weights (either uniformly or globally) and then prune gradually every 10 epochs until epoch 80, at which point we fine-tune for a further 20 epochs.

The results are given in Figure 6. In terms of accuracy, Global-GMP outperforms Uniform-GMP at the same target sparsity, with Global-95% showing similar accuracy to Uniform-90%, though in all cases performance is inferior to when ImageNet weights are used for pre-training. The highest speedup is of 1.31x, for 95% Uniform sparsity.

Additionally, we evaluate SparseProp on the CelebA dataset (Liu et al., 2015), which consists of a training set of 162'770 images, and a validation set of 19'962 images from 10 000 well-known individuals, each annotated with forty binary attributes, such as "Smiling", "Male", "Wearing Necklace", etc. We consider the task of jointly predicting all forty attributes, in a similar setup as above, and show the resulting AUC in Table 5. We observe that AUC stays fairly constant even at high sparsities, even as the speed of training increases.

Finally, we examine the performance of SparseProp on the RigL method (Evci et al., 2020), a dynamic sparse training technique, to train a sparse ResNet18 model. Our evaluation is conducted on the standard CIFAR-10 dataset (Krizhevsky et al.). We train the model for 100 epochs with a batch size of 256. We set the hyper-parameters $\alpha = 0.3$, $T_{end} = 80$, and $\Delta T = 10$ (introduced in the original paper (Evci et al., 2020)). The results, summarized in Table 6, demonstrate the achieved speedups and accuracy drops when compared to the dense implementation. Notably, the speedups attained through RigL surpass those obtained through GMP. This is because RigL initializes the network layers with the target sparsity at the outset of training, while GMP progressively achieves the target sparsity during training.

Language Modelling. For sparse fine-tuning from scratch on language models, we start from the pre-trained BERT-base (Devlin et al., 2019) model, which we fine-tune for 3 epochs on the target downstream task, and then prune in *one-shot* with the state-of-the-art unstructured pruner oBERT (Kurtic et al., 2022), uniformly to 90%, 95% or 97% per-layer sparsity. After one-shot pruning, we fine-tune the remaining weights for 5 epochs and examine accuracy and speedup versus the dense variant. The results are presented in Table 5 (right), and show end-to-end speedups of up to 30%, at an accuracy loss between 1 and 3.5%.

5. Discussion

We have provided an efficient vectorized algorithm for sparse backpropagation, with linear runtime dependency in the density of the layer weights. We have also provided an efficient CPU-based implementation of this algorithm, and integrated it with the popular Pytorch framework. Experimental evidence validates the runtime scaling of our algorithm on various layer shapes and types. We complemented this algorithmic contribution with an extensive study of the feasibility of sparse transfer learning and from-scratch training in edge scenarios. We observed consistent speedups across scenarios, at the cost of moderate accuracy loss. Our results should serve as motivation for further research into accurate sparse training in this setting, in particular for leveraging sparsity on highly-specialized tasks, which is an under-studied area.

There are several promising directions for future research to extend SparseProp. One potential direction is to enable SparseProp to leverage GPUs. To support this possibility, one could leverage the Sputnik library (Gale et al., 2020b), which showcases reasonable speedups for SpGEMM and SDDMM operations with unstructured sparsity. Another potential direction for expanding the capabilities of SparseProp is to enable it to handle different types of sparsity patterns, such as the recent N:M format. Implementing support for these patterns would require making adjustments to the way matrix multiplications are performed. For example, in the N:M scenario, it would be necessary to store two matrices for the weights: one containing the non-zero values and another indicating the corresponding non-zero indices in each block.

6. Acknowledgments

We would like to thank Elias Frantar for his valuable assistance and support at the outset of this project, and the anonymous ICML and SNN reviewers for very constructive feedback. EI was supported in part by the FWF DK VGSCO, grant agreement number W1260-N35. DA acknowledges generous ERC support, via Starting Grant 805223 ScaleML.

References

Berg, T., Liu, J., Lee, S. W., Alexander, M. L., Jacobs, D. W., and Belhumeur, P. N. Birdsnap: Large-scale fine-grained visual categorization of birds. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

Bossard, L., Guillaumin, M., and Van Gool, L. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision (ECCV)*, 2014.

Chen, T., Frankle, J., Chang, S., Liu, S., Zhang, Y., Carbin, M., and Wang, Z. The lottery tickets hypothesis for supervised and self-supervised pre-training in computer vision models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16306–16316, 2021.

Cimpoi, M., Maji, S., Kokkinos, I., Mohamed, S., and Vedaldi, A. Describing textures in the wild. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

Dettmers, T. and Zettlemoyer, L. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

Elsen, E., Dukhan, M., Gale, T., and Simonyan, K. Fast sparse convnets. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning (ICML)*, 2020.

Gale, T., Zaharia, M., Young, C., and Elsen, E. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020a.

Gale, T., Zaharia, M., Young, C., and Elsen, E. Sparse GPU kernels for deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020b.

Gray, S., Radford, A., and Kingma, D. P. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3:2, 2017.

Griffin, G., Holub, A. D., and Perona, P. The Caltech 256. *Caltech Technical Report*, 2006.

Hagiwara, M. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207 – 218, 1994. ISSN 0925-2312. Backpropagation, Part IV.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016a.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016b.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

Hubara, I., Chmiel, B., Island, M., Banner, R., Naor, S., and Soudry, D. Accelerated sparse neural training: A provable and efficient method to find N:M transposable masks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

Iofinova, E., Peste, A., Kurtz, M., and Alistarh, D. How well do sparse ImageNet models transfer? In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

Ivanov, A., Dryden, N., and Hoefler, T. Sten: An interface for efficient sparsity in pytorch. 2022.

Jakob, W., Rhinelander, J., and Moldovan, D. pybind11 — seamless operability between c++11 and python, 2016. URL `https://github.com/pybind/pybind11`.

Jayakumar, S., Pascanu, R., Rae, J., Osindero, S., and Elsen, E. Top-KAST: Top-K always sparse training. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Jiang, P., Hu, L., and Song, S. Exposing and exploiting fine-grained block structures for fast and accurate sparse training. In *Advances in Neural Information Processing Systems*, 2022.

Kornblith, S., Shlens, J., and Le, Q. V. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2661–2671, 2019.

Krause, J., Stark, M., Deng, J., and Fei-Fei, L. 3D Object Representations for Fine-Grained Categorization. In *4th International IEEE Workshop on 3D Representation and Recognition*, Sydney, Australia, 2013.

Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research). URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Kurtic, E., Campos, D., Nguyen, T., Frantar, E., Kurtz, M., Fineran, B., Goin, M., and Alistarh, D. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4163—4181, 2022.

Kusupati, A., Ramanujan, V., Somani, R., Wortsman, M., Jain, P., Kakade, S., and Farhadi, A. Soft threshold weight reparameterization for learnable sparsity. In *International Conference on Machine Learning (ICML)*, 2020.

Lagunas, F., Charlaix, E., Sanh, V., and Rush, A. M. Block pruning for faster transformers. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *Conference on Neural Information Processing Systems (NeurIPS)*, 1990.

Lhoest, Q., Villanova del Moral, A., Jernite, Y., Thakur, A., von Platen, P., Patil, S., Chaumond, J., Drame, M., Plu, J., Tunstall, L., Davison, J., Šaško, M., Chhablani, G., Malik, B., Brandeis, S., Le Scao, T., Sanh, V., Xu, C., Patry, N., McMillan-Major, A., Schmid, P., Gugger, S., Delangue, C., Matussière, T., Debut, L., Bekman, S., Cistac, P., Goehringer, T., Mustar, V., Lagunas, F., Rush, A., and Wolf, T. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 175–184. Association for Computational Linguistics, November 2021.

Li, F.-F., Fergus, R., and Perona, P. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.

Lis, M., Golub, M., and Lemieux, G. Full deep neural network training on a pruned weight budget. *Proceedings of Machine Learning and Systems*, 1:252–263, 2019.

Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

Maji, S., Rahtu, E., Kannala, J., Blaschko, M., and Vedaldi, A. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

NeuralMagic. DeepSparse, 2022. URL `https://github.com/neuralmagic/deepsparse`.

Nilsback, M.-E. and Zisserman, A. A visual vocabulary for flower classification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006.

NVIDIA. Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT, 2021. URL `https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/`.

Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., and Dubey, P. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

Parkhi, O. M., Vedaldi, A., Zisserman, A., and Jawahar, C. V. Cats and dogs. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*. 2019a.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019b.

Peste, A., Iofinova, E., Vladu, A., and Alistarh, D. AC/DC: Alternating compressed/decompressed training of deep neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

Raihan, M. A. and Aamodt, T. Sparse weight activation training. *Advances in Neural Information Processing Systems*, 33:15625–15638, 2020.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3): 211–252, 2015.

Salman, H., Ilyas, A., Engstrom, L., Kapoor, A., and Madry, A. Do adversarially robust ImageNet models transfer better? *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Sanh, V., Wolf, T., and Rush, A. M. Movement pruning: Adaptive sparsity by fine-tuning. *arXiv preprint arXiv:2005.07683*, 2020.

Schwarz, J., Jayakumar, S., Pascanu, R., Latham, P., and Teh, Y. Powerpropagation: A sparsity inducing weight reparameterisation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

Singh, S. P. and Alistarh, D. WoodFisher: Efficient second-order approximation for neural network compression. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

SparseZoo, N. DeepSparse, 2022. URL `https://github.com/neuralmagic/sparsezoo`.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

Wang, Z. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. *arXiv preprint arXiv:2008.11849*, 2020.

Wiedemann, S., Mehari, T., Kepp, K., and Samek, W. Dithered backprop: A sparse and quantized backpropagation algorithm for more efficient deep neural network training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 720–721, 2020.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Xiao, J., Hays, J., Ehinger, K., Oliva, A., and Torralba, A. Sun database: Large-scale scene recognition from abbey to zoo. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.

Yang, D., Ghasemazar, A., Ren, X., Golub, M., Lemieux, G., and Lis, M. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 711–724. IEEE, 2020.

Zafrir, O., Larey, A., Boudoukh, G., Shen, H., and Wasserblat, M. Prune once for all: Sparse pre-trained language models. *arXiv preprint arXiv:2111.05754*, 2021.

Zhang, Z., Yang, P., Ren, X., Su, Q., and Sun, X. Memorized sparse backpropagation. *Neurocomputing*, 415:397–407, 2020.

Zhou, X., Zhang, W., Chen, Z., Diao, S., and Zhang, T. Efficient neural network training via forward and backward propagation sparsification. *Advances in Neural Information Processing Systems*, 34:15216–15229, 2021.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

A. Additional Detailed Results

In this section, we describe the twelve datasets we use to train image classification models in sections 4.2.1 and 4.2.2, as well as present the complete per-dataset accuracy results for transfer and from-scratch training on these datasets.

Dataset	Number of Classes	Train/Test Examples	Accuracy Metric
SUN397(Xiao et al., 2010)	397	19 850 / 19 850	Top-1
FGVC Aircraft(Maji et al., 2013)	100	6 667 / 3 333	Mean Per-Class
Birdsnap(Berg et al., 2014)	500	32 677 / 8 171	Top-1
Caltech-101(Li et al., 2004)	101	3 030 / 5 647	Mean Per-Class
Caltech-256(Griffin et al., 2006)	257	15 420 / 15 187	Mean Per-Class
Stanford Cars(Krause et al., 2013)	196	8 144 / 8 041	Top-1
CIFAR-10(Krizhevsky et al., 2009)	10	50 000 / 10 000	Top-1
CIFAR-100(Krizhevsky et al., 2009)	100	50 000 / 10 000	Top-1
Describable Textures (DTD)(Cimpoi et al., 2014)	47	3 760 / 1 880	Top-1
Oxford 102 Flowers(Nilsback & Zisserman, 2006)	102	2 040 / 6 149	Mean Per-Class
Food-101(Bossard et al., 2014)	101	75 750 / 25 250	Top-1
Oxford-IIIT Pets(Parkhi et al., 2012)	37	3 680 / 3 669	Mean Per-Class

Table 7. Target tasks for from-scratch and transfer learning.

Dataset	Dense	Uniform 90%	Uniform 97%	Global 95%
Aircraft	83.6 ± 0.4	81.4 ± 0.3	79.0 ± 0.0	81.2 ± 0.4
Birds	72.4 ± 0.3	68.7 ± 0.1	67.8 ± 0.0	66.9 ± 0.1
CIFAR-10	97.4 ± 0.0	97.0 ± 0.0	96.7 ± 0.3	96.2 ± 0.1
CIFAR-100	85.6 ± 0.2	84.5 ± 0.1	84.0 ± 0.1	82.9 ± 0.1
Caltech-101	93.5 ± 0.1	92.5 ± 0.1	92.1 ± 0.3	91.9 ± 0.2
Caltech-256	86.1 ± 0.1	85.1 ± 0.0	83.6 ± 0.0	83.1 ± 0.0
Cars	90.3 ± 0.2	88.2 ± 0.2	87.0 ± 0.1	87.6 ± 0.1
DTD	76.2 ± 0.3	75.1 ± 0.0	74.8 ± 0.2	74.1 ± 0.4
Flowers	95.0 ± 0.1	95.0 ± 0.0	95.3 ± 0.4	94.1 ± 0.3
Food-101	87.3 ± 0.1	86.5 ± 0.1	85.7 ± 0.0	85.5 ± 0.0
Pets	93.4 ± 0.1	92.3 ± 0.1	90.1 ± 0.0	91.0 ± 0.1
SUN397	64.8 ± 0.0	63.4 ± 0.0	62.4 ± 0.1	61.4 ± 0.2

Table 8. Transfer accuracy for sparse ResNet50 models pretrained on ImageNet1K.

Dataset	Dense	Uniform 90 %	Uniform 95%	Global 95%
Aircraft	58.1 ± 0.3	56.1 ± 0.4	56.0 ± 0.1	57.6 ± 0.3
Birds	51.5 ± 0.7	50.3 ± 0.3	49.2 ± 0.5	50.1 ± 0.3
CIFAR-10	94.3 ± 0.3	93.7 ± 0.1	93.1 ± 0.2	93.5 ± 0.1
CIFAR-100	74.6 ± 0.0	73.4 ± 0.3	72.8 ± 0.3	72.8 ± 0.6
Caltech-101	46.4 ± 0.9	45.7	45.0	45.6 ± 0.8
Caltech-256	47.1 ± 0.8	46.2 ± 0.8	45.7 ± 0.7	46.4 ± 0.8
Cars	67.1 ± 0.1	63.8 ± 0.5	62.7 ± 0.8	64.5 ± 0.5
DTD	37.8 ± 1.3	36.3 ± 1.5	37.0 ± 1.0	37.7 ± 1.1
Flowers	59.3 ± 0.1	59.4 ± 0.5	58.5 ± 0.8	58.2 ± 0.4
Food-101	78.3 ± 1.7	77.1 ± 0.0	76.0 ± 0.2	77.4 ± 0.3
Pets	59.2 ± 0.2	58.7 ± 0.3	57.3 ± 0.6	59.1 ± 0.6
SUN397	40.5 ± 3.2	40.0 ± 0.4	39.5 ± 0.1	39.5 ± 0.1

Table 9. From-scratch training accuracy for sparse ResNet18 Models trained on standard training datasets.