Inducing and Exploiting Activation Sparsity for Fast Neural Network Inference

Mark Kurtz [*1] **Justin Kopinsky** [*1] **Rati Gelashvili** [1] **Alexander Matveev** [1] **John Carr** [1] **Michael Goin** [1] **William Leiserson** [1] **Sage Moore** [1] **Bill Nell** [1] **Nir Shavit** [1] **Dan Alistarh** [1 2]

Abstract

Optimizing deep neural networks for inference has recently become an extremely active area of research. One of the go-to solutions in this context is weight pruning, which aims to reduce computational and memory footprint by removing large subsets of the connections in a neural network. Surprisingly, much less attention has been given to exploiting sparsity in the *activation maps*, which tend to be naturally sparse in many settings thanks to the structure of rectified linear (ReLU) activation functions. In this paper, we present an analysis of methods for maximizing the sparsity of the activations in a trained neural network, and show that, when coupled with an efficient sparse-input convolution algorithm, we can leverage this sparsity for significant performance gains. To induce highly sparse activation maps without accuracy loss, we introduce a new regularization technique, coupled with a new threshold-based sparsification method based on a parameterized activation function called Forced-Activation-Threshold Rectified Linear Unit (FATReLU). We examine the impact of our methods on popular image classification models, showing that most architectures can adapt to significantly sparser activation maps without any accuracy loss. Our second contribution is showing that these these compression gains can be translated into inference speedups: we provide a new algorithm to enable fast convolution operations over networks with sparse activations, and show that it can enable significant speedups for end-to-end inference on a range of popular models on the large-scale ImageNet image classification task on modern Intel CPUs, with relatively low retraining cost.

[*]Equal contribution [1]Neural Magic [2]IST Austria. Correspondence to: Dan Alistarh <dan@neuralmagic.com>.

Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

1. Introduction

Deep neural networks (DNNs) are able to achieve state-of-the-art performance in several application domains, such as image classification, speech recognition, and automated decision making, e.g. (Krizhevsky et al., 2012; Vaswani et al., 2017; Silver et al., 2016). Along with this wide array of applications comes the need to reduce the significant computational and memory footprint of DNNs. To this end, several techniques have been designed to obtain optimized, resource-efficient variants of a given deep model. *Pruning* and *quantization* are arguably the standard methods for achieving resource-efficient models, which have received considerable attention, e.g. (Liu et al., 2017; Luo et al., 2017; Gray et al., 2017; Han et al., 2015; Li et al., 2016; Mishra et al., 2017; Zhu et al., 2016). However, the vast majority of existing work has focused on compressing the *weights (connections)* in the neural network, for which several regularization (Molchanov et al., 2017) and thresholding-based methods (Han et al., 2015; Gale et al., 2019) are now known.

It is therefore perhaps surprising that sparsifying *activation maps* has received relatively little attention. A non-trivial fraction of the activations are sparse as a natural consequence of the structure of Rectified Linear Unit (ReLU) activation functions. This observation has been leveraged by hardware accelerators, e.g. (Albericio et al., 2016; Han et al., 2016; Parashar et al., 2017), and reference (Rhu et al., 2018) performed an analysis of naturally-occurring activation sparsity. Recently, (Georgiadis, 2019) explored *L1 regularization* to increase the number of zeroes in the activation maps, showing that sparsity can be increased by up to 60% for image classification models.

A second gap in the literature is the absence of software support for sparsity, and in particular *activation* sparsity, on common hardware. Currently, running models with higher activation sparsity rates on common CPU or GPU platforms will not result in computational speedups, and improvements are only reported in relative sparsity percentage, or synthetic memory compression rates (Georgiadis, 2019). It is not at all clear how these compression rates will relate to speedup in real-world implementations, and it is therefore difficult to evaluate the practical impact of existing methods.

In this paper, we address both these gaps with respect to activation sparsity. We begin by performing an in-depth analysis of *regularization and thresholding methods* as a way to increase activation map sparsity in convolutional neural networks. Specifically, we present a set of techniques which can significantly boost naturally-occurring activation sparsity in CNNs, without loss of accuracy. Our methods can be both applied *statically* (requiring no retraining) and *dynamically* (if fine-tuning is possible), and significantly improve upon existing regularization-based methods (Georgiadis, 2019), often by more than $2\times$ in terms of relative improvement to baseline sparsity. We complement these techniques with negative results, showing that activation sparsification cannot smoothly recover accuracy through re-training (as opposed to weight sparsification (Gale et al., 2019)), and that applying thresholding independently *per each channel* is possible but only yields limited gains. Our second contribution is a general algorithm which can leverage activation sparsity for computational gains, and its efficient CPU-based implementation. The resulting framework can lead to inference speedups of more than 2x on a range of popular CNNs for image classification, relative to industrial CPU- and GPU-based inference frameworks, and to our optimized dense baseline.

Our sparsity-boosting methods combine a regularizer following the Hoyer sparsity metric (Hoyer, 2004), together with a variant of the classic ReLU activation, which we call Forced Activation Threshold ReLU (FATReLU). Simply put, FATReLU implements a variable threshold for the common ReLU activation function, below which all activations are set to zero, based on the intuition is that a non-trivial fraction of the positive activations can be eliminated without significant impact on the output. We develop techniques to determine and optimize FATReLU thresholds per layer, and perform an analysis of the interplay between these methods and the accuracy of the resulting model. In short, we find that sparsity can be significantly boosted via Hoyer regularization and thresholding, with no accuracy loss, beyond L1 regularization. The methods we propose induce negligible ($< 0.3\%$) accuracy loss on ImageNet-scale models, and can even result in minor accuracy increase. However, contrary to weight pruning methods, which can gradually trade off accuracy for increased sparsity, we find that *sharp* thresholds exist for activations, beyond which accuracy drops, and cannot be recovered. This observation simplifies the fine-tuning process, since, up to this threshold, we are usually able to recover full accuracy, and there is little benefit in fine-tuning beyond this threshold.

Our second contribution is a computational framework to leverage activation sparsity for computational gains, tailored to CPUs. This framework is based on an algorithm for fast convolutions on sparse inputs, for which we present an efficient vectorized implementation, and back by several non-trivial optimizations. We implement our framework in C++, and test it on a range of popular CNNs for image classification on the classic ImageNet ILSVRC2012 dataset (Deng et al., 2009). We find that 1) many popular models have significant "natural" activation sparsity, without any specific activation regularization; 2) the natural activation sparsity of these networks can be consistently and significantly boosted using our techniques. We show the resulting boosted models can be executed with speedups of more than $2\times$ compared to state-of-the-art CPU and GPU inferencing solutions.

Related Work. The literature on model compression for DNNs is extremely vast, so we restrict our attention to work on analyzing and leveraging activation sparsity. The fact that activation sparsity arises naturally is well-known, and has been leveraged by several architecture proposals, e.g. (Albericio et al., 2016; Han et al., 2016; Parashar et al., 2017); in particular, reference (Rhu et al., 2018) performed an in-depth analysis of activation sparsity on a range of convolutional models. We extend this analysis here.

Another related line of work is that on compressing activation maps. A common technique for reducing the memory footprint of activation maps is *quantization*, which has been employed successfully by several references, see e.g. (Mishra et al., 2017) and references therein. We do not investigate quantization here, and leave a thorough treatment of the impact of our sparsification techniques in conjunction with quantization for future work. Reference (Gudovskiy et al., 2018) proposed a projection technique coupled with non-linear dimensionality reduction, which required modifying the network structure, while (Alwani et al., 2016) proposed to *stochastically* prune activations as an adversarial defense. Both techniques cause significant accuracy loss, and are therefore outside the scope of our study. Agostinelli et al. (2014) propose learning piecewise linear activation functions to improve the accuracy of given models. FATReLU is piecewise linear, but the goals and methods we investigate in this paper are different.

The work closest to ours is (Georgiadis, 2019), who proposed and investigated the use of L1-regularization applied to the activation maps, and showed that it can result in a significant increase (up to 60% relative to naturally-occurring activation sparsity) on a range of CNNs for image classification. The paper goes on to explore several efficient encoding techniques for the activations, and evaluates them synthetically in terms of their resulting compression factors, but provides no inference experiments. We show that Hoyer regularization is superior to L1, in the sense that it provides higher activation sparsity without accuracy loss on all the models we investigated. The thresholding methods we propose are complementary to regularization in the sense that they can be applied independently of whether the base model has been regularized or not, or of the regularization method. In addition, our paper provides a complete framework for leveraging activation sparsity for fast inference on CPUs, as well as end-to-end inference speedup for activation-sparsified models.

To our knowledge, the only reference to explicitly leverage input sparsity for performance gains is the recent preliminary publication of (Dong et al., 2019). By contrast, their algorithm is more complex, and requires high input sparsity to be efficient: in particular, as stated in the reference, the resulting algorithm can only be applied to certain types of tasks and models, such as LiDAR-based detection, or character recognition. For this reason, we do not directly compare against it. Our technique is applicable and efficient in a much wider range of scenarios.

Related algorithmic ideas have been investigated in (Park et al., 2016b;a; Chen, 2018). The critical distinction is that all these references explore leveraging sparsity in the *weights*, rather than in the *activations*, leading to a different algorithm structure and implementation. For example, our procedure critically requires efficient on-the-fly input compression, whereas weight sparsity techniques can precompress the weights offline. Another key difference from these approaches is that they *require retraining*, since kernel sparsity is not naturally present in neural networks without specific regularization or thresholding. Second, the speedups achieved by these methods are bound to be limited by the fact that, even with thresholding, kernels cannot usually be sparsified to the extremely large ratios which can be naturally present in activations without loss, e.g $> 90\%$.

Our work can also be examined in the broader context of model compression methods, which is an extremely active research area, e.g. (Wu et al., 2016; Zhu et al., 2016; Mishra et al., 2017; Mellempudi et al., 2017; Zhang et al., 2017; Park et al., 2016b; Han et al., 2016; Polino et al., 2018; Frankle & Carbin, 2018). We develop the first thresholding-based method specifically for *activations*, along with specific sensitivity analysis and tuning techniques.

2. Activation Sparsity in CNNs

2.1. Natural and Regularized Activation Sparsity

Naturally-Arising Sparse Activations. We begin by examining the natural sparsity of activation maps in CNNs. For simplicity, we will focus on residual models trained on the ImageNet (ILSVRC2012) task, although our findings are generally valid across other datasets (in particular, CIFAR-10 and 100 (Krizhevsky et al., 2014)) and architectures (ResNet (He et al., 2016), Mobilenet (Howard et al., 2017))–please see Section 5 for full results.

Activation sparsity is linked with the structure of the ReLU non-linearity: if input data to this function were completely random and zero-centered, then we would expect an output activation sparsity concentrated around 50%. However, if we examine the average activation map sparsity across several batches, we notice that layers which are closer to the input tend to have activation sparsity that is *lower* than this threshold, whereas later layers tend to have *higher* activation sparsity. One intuitive (but imprecise) explanation for this phenomenon could be that earlier layers adapt to extract more numerous low-level features, whereas the later layers would extract higher-level features. Please see Figure 4 for an illustration. The standard deviation of the recorded sparsities is under 1% across batches, so we omit confidence intervals for visibility, noting that this stable behaviour across batches is somewhat surprising.

The Impact of Network Depth and Width. In this context, it is natural to ask whether wider or deeper networks will tend to have higher activation sparsity. We examined this trend on pre-trained ImageNet models, in particular comparing ResNet50 with its 2x wide variant (Zagoruyko & Komodakis, 2016), as well as with the deeper ResNet101 and its 2x wide variant. We use the Torchvision pretrained models as examples. The results are provided in Table 3. (We observed similar results in a depth-width ablation study on residual networks on CIFAR-10, which we omit for brevity.) First, average activation sparsity does indeed increase with network depth (e.g. 53% to 57% for ResNet50 vs ResNet101), corroborating the intuition that "higher level features" develop deeper in the network. Second, wider networks do have a higher fraction of zero activations (e.g. 53% to 58% for ResNet50 vs 2xWideResNet50), matching the intuition that only a limited subset of the features are necessary to classify a certain input, whose proportion does not necessarily increase with layer width. Moreover, as can be seen from the result for 2xWide ResNet101 (63%), these trends compound.

L1 Regularization. In Figure 4(a), we also examine the impact of L1 regularization applied to the activations on the sparsity. We follow the proposal of (Georgiadis, 2019), which consists of fine-tuning an accurate pre-trained model with L1 regularization for a number of epochs, as well as the carefully optimized regularization parameter values provided, which ensure no accuracy loss. We notice that this method can boost the sparsity of activations by an extra 1% and 4% on average on ResNet50 and Mobilenet, respectively. (See Table 1 for values across models.)

Hoyer Regularization. We go beyond the L1 sparsity-inducing regularization, and consider the *square Hoyer* regularizer, defined for a vector $\vec{v}$ as $H(\vec{v}) = \frac{\left(\sum_{i=1}^{d} |v_i|\right)^2}{\sum_{i=1}^{d} v_i^2}$. This regularizer has a range of desirable properties as a measure of sparsity (Hoyer, 2004), such as scale-invariance and differentiability almost-everywhere. It is popular for compressed sensing, and has only recently been applied for *weight* sparsification (Yang et al., 2019); to our knowledge, we are the first to investigate it for activation sparsity.

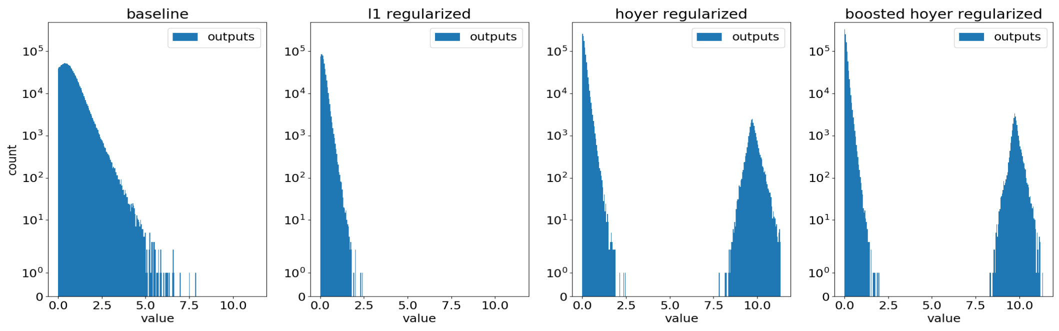


Figure 1. Illustration of the impact of regularization and boosting on the output distribution of a convolutional layer (ResNet18, layer 5). The Y axis is log-scale. Notice that all methods significantly narrow the set of non-zero activations; however, Hoyer and boosted Hoyer allow for more "diversity" in the activations, which explains their better performance.

Figure 4(a) presents the output activation sparsities for each layer of ResNet18, when regularized with square Hoyer such that there is no accuracy loss. Specifically, for each ReLU's output we apply the square Hoyer regularization multiplied by a hyperparameter determined experimentally to the cost function. We found values between 10^{-8} (conservative) to 10^{-7} (more aggressive) to work for this parameter, for all the models we considered. Our initial learning rate for retraining is 5×10^{-3}, and we maintain standard momentum and weight decay values. With these parameters, we retrain for 10 epochs to stabilize weights and recover accuracy. We note that this recalibration process is significantly less expensive than for L1 regularization (Georgiadis, 2019), which required 90 epochs of training for recovery. The improvements relative to the additional sparsity induced by L1 are of 2.4x and 8x, for Mobilenet and ResNet50, respectively. Our experimental results in Section 5 clearly suggest that square Hoyer is superior to classic L1 regularization.

2.2. The Distribution of Activations

We now focus our attention to the *distribution* of activations in the layers of a neural network. We performed a basic histogram analysis for layers of ResNet18, from the original pre-trained model, as well as from the L1, Hoyer-regularized, and boosted variants of the same model. We notice that, for all instances, a non-trivial fraction of the activations are clustered around zero. Next, we implement an *activation sensitivity analysis* procedure: independently for each layer, we fix a threshold T below which all of the activations will be set to zero. We then increase this threshold and examine the loss of accuracy. The resulting graph for a set of layers of pretrained ResNet18 is presented in Figure 2. Results suggest that a non-trivial fraction of the activations can be set to zero without affecting the loss. The results presented are averaged over a set of 128 mini-batches. We found these results to be extremely consistent, and therefore omit error bars for visibility.

Further, Figure 2 (center, right), shows that regularization may serve to *stabilize* activations, in the sense that a larger fraction can be thresholded on regularized models, without accuracy loss. Moreover, we found the benefits from regularization to be approximately independent from, and additive with, the benefits from thresholding. Layers other than the one depicted exhibited a similar pattern, with some variance in the particular sparsity values.

3. Boosting Activation Sparsity

In this section, we investigate generic ways to systematically produce networks with high activation sparsity. We begin with *static* methods (which require no retraining), and then continue with *dynamic* methods, which are allowed to retrain in order to recover accuracy.

Forced-Activation Thresholds. Formally, the Forced-Activation Threshold ReLU activation function (FATReLU) is simply defined as:

$$FATReLU_T(x) = \begin{cases} x & \text{when } x \geq T \text{ ;} \\ 0 & \text{otherwise.} \end{cases}$$

Note that FATReLU cannot be simulated by simply adding a linear bias term to ReLU. Further, not only is FATReLU not differentiable at T, but it is not even *continuous* at T, which renders training neural networks from scratch with FATReLU cumbersome. However, our use case allows us to use it to directly replace ReLU on a pre-trained model whose activations we wish to further sparsify.

Baseline Model. We assume an accurate pre-trained model for the target architecture and task. We first fine-tune the provided model using the square Hoyer regularizer, which sets a fraction of the activations to zero, and also "stabilizes" the other activations, allowing a larger fraction to be thresholded via FATReLU.

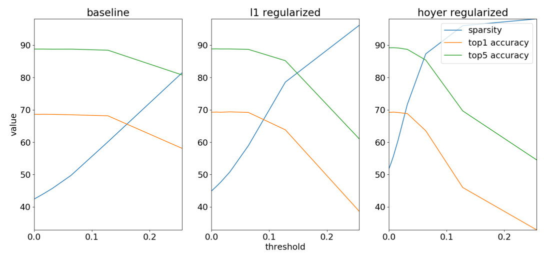


Figure 2. Illustration of the sensitivity analysis results for a single convolutional layer in ResNet18, for different threshold values (X axis) and different regularizers (panels).

Activation Sensitivity Analysis. We first aim to find layer-wise activation thresholds which sparsify a large fraction of the activations preserving accuracy. We adapt the weight sparsity sensitivity analysis (Han et al., 2015) for the case of activations. Intuitively, we estimate the "derivative" of the loss with respect to the activation sparsity of each layer. The procedure starts by identifying a set of target sparsity percentages for the outputs of the different layers. For each layer L, we pick a maximal percentage T_L of the *extra* activations which should be set to zero, in addition to the natural sparsity. We evaluate and record the loss at discrete thresholds $t \in [0, T_L]$. (This procedure exclusively uses batches from the *training set*.)

We thus obtain a "sensitivity profile" for each layer, based on which we set a threshold for the activations of the layer. We usually pick the threshold for each layer to be the largest value which did not result in accuracy loss, modulo some small error tolerance. A typical set of results is illustrated in Figure 2. It is not uncommon for $FATReLU_t$ to *improve* accuracy at low threshold values– one possible explanation is that this serves to remove some of the noise from the activations close to the zero threshold.

Retraining and Sharp Activation Thresholds. The above procedure is *static*, in the sense that the model weights are not modified, and the model is not retrained. It results in consistent, but relatively limited improvements in terms of activation sparsity: for instance, for the ResNet18 model, the average increase across layers due to static boosting is under 3% globally. We wish to achieve higher thresholds by allowing *retraining* of the network to adapt the weights to the higher thresholds. Such procedures are common for weight sparsification (Zhu & Gupta, 2017; Gale et al., 2019).

Perhaps surprisingly, we find that this behavior is *bi-modal* for activations: we can increase activation sparsity within a continuous range and still have the model recover full accuracy through retraining at each level within the range. However, each layer appears to have a "sharp" activation threshold beyond which the model is no longer able to recover accuracy, even with significant retraining. Identifying the exact root cause of this phenomenon is difficult, but we conjecture that it is related to the fact that the forward and backward information flow through the layer is break down due to the high activation threshold.

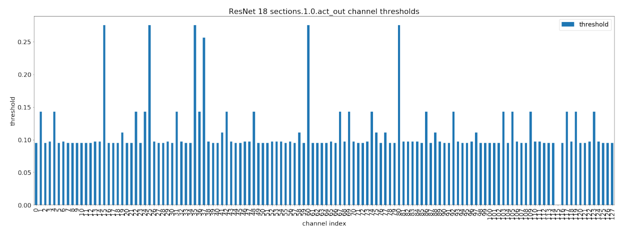
Dynamic Thresholding. Due to this bi-modal recovery behavior, we use dynamic thresholding to simplify the process of finding the optimal thresholds per layer. We fix a small accuracy loss tolerance, τ (0.2% in our experiments), and, for each layer, we refer to the static analysis results to identify the maximal threshold for which accuracy loss remained below τ, determined by binary search over the range of thresholds. Once this threshold is determined for each layer, we run *one* fine-tuning training epoch until either recovery is achieved, or recovery fails. Using this success or failure criterion, we can perform binary search on τ to determine the largest τ for which recovery is possible.

We adopt this procedure since it has low cost, and similar outcomes to more complex iterative procedures we have investigated, both in terms of sparsity and accuracy. In addition, Dynamic Thresholding performs particularly well when used in conjunction with Hoyer regularization (please see Figure 4(a)). We adopt Hoyer regularization plus Boosting via Dynamic Thresholding as our main method for generating activation-sparse models.

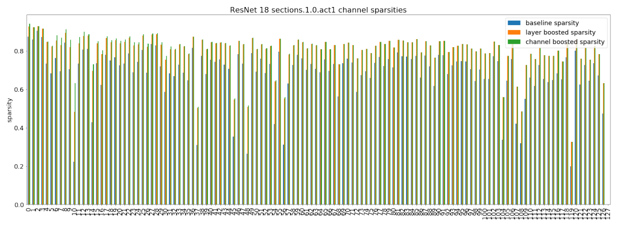
Channel-wise Thresholding. Next, we ask whether we could further increase activation sparsity by performing Dynamic Thresholding *channel-wise*, setting a distinct threshold for each channel of each layer. This procedure is costly, since it requires fine-grained tuning across each channel, and requires care, since the impact of each individual channel on the loss may be small. We proceed as follows.

We start from the layer-wise FATReLU thresholds determined above. Next, we perform a one-shot sensitivity analysis for each channel in each layer, by estimating the piece-wise integral of the cross-entropy loss relative to the channel threshold, obtained from sensitivity analysis. We adopt the maximum threshold across all channels as the maximum value to integrate for across all channels. A lower integral value suggests that the channel is less sensitive to thresholding. Based on the results of this channel-wise sensitivity procedure, we partition the channels into groups based on their sensitivity. For each channel group (e.g. the 25% least sensitive channels, and so on), we perform binary search on their joint thresholds, attempting to *increase* their FATReLU threshold, until the point where we reach the tolerance in terms of accuracy difference.

We have performed channel-wise thresholding for ResNet18 following the above procedure. Please see Figure 3 for a sample of the results. On the positive side, the procedure does not diverge–we are able to systematically increase thresholds per channels without accuracy loss. On the nega-



(a) Channel Thresholds after Channel-Wise Sensitivity Analysis. Notice the relatively low proportion of channels which can be boosted past the average threshold.



(b) Sparsity Boosting across Channels. We can only obtain a low average activation sparsity increase, at significantly increased computational cost.

Figure 3. Channel-wise boosting results for a fixed "highly-sensitive" layer of ResNet18.

tive side, we can only obtain an average activation sparsity increase of approximately 2% relative to the coarse-grained dynamic thresholding method, at significantly increased computational cost. Further analysis of the results suggests that a fraction of approximately 30% of the channels cannot be boosted past the layer threshold, whereas a small fraction of approximately 10% of the channels have negligible impact on the loss and thus can be completely eliminated. The cost of this method outweigh its computational benefits.

4. Leveraging Activation Sparsity

Background. To make use of activation sparsity at runtime, we implement an algorithm to perform sparse convolutions on data that is initially produced (e.g. from a previous layer) in a standard (i.e. dense) format. We make use of a variant of the *Compressed Sparse Row* (CSR) representation (e.g. as implemented in (Wang et al., 2014)). Prior work has taken advantage of CSR for computing convolutions when the *kernels* are sparse, on both GPUs (Park et al., 2016b) and CPUs (Park et al., 2016a), where one has the luxury of being able to *pre-compress* the sparse kernels prior to inference with no performance overhead. However, for activations, the location of the non-zero elements is not known until inference time, and so we must be able to efficiently compress the activations at run time. Once compressed, we can apply Algorithm 1 to the compressed input. Importantly, both CSR compression and sparse-input convolution can be implemented efficiently on modern hardware, i.e. without the need to branch on zero elements.

We use a "3-array" variation of CSR, wherein a sparse matrix M is represented with the following three arrays:

- `values`: Element j contains the j^{th} non-zero element of M in row-major order
- `columns`: Element j contains the column index in M of the corresponding element `values`$[j]$
- `row_pointers`: Element i contains a pointer to the first element in `values` which came from row i of M

Note that `row_pointers` serves the additional function of encoding the number of non-zero elements per row, i, derivable as `row_pointers`$[i+1]$ $-$ `row_pointers`$[i]$.

The Algorithm. Algorithm 1 shows a simple pseudo-code implementation to compute the convolution of a dense kernel K with sparse input I given in CSR format to produce output O. For simplicity, we assume that the input data has one channel dimension and one spatial dimension. In particular, the input I is a CSR representation of data with dimensionality $I_C \times I_x$, the output O is a matrix with dimensions $O_C \times O_x$, and the kernel K is a tensor with dimensions $O_C \times I_C \times K_x$. Extending to more spatial dimensions (as is typical, e.g., in image processing NNs) is straightforward and omitted for clarity.

AVX Implementation. We implemented Algorithm 1 on Intel's Skylake architecture with x86+AVX512 instruction set. Both CSR compression and Algorithm 1 can be implemented efficiently using available SIMD instructions. Algorithm 2 demonstrates how to implement Algorithm 1 in a SIMD way. Note that $\hat{O}_C$ refers to the number of *vectors* of output channels to be computed, i.e. $\hat{O}_C = O_C/r$ when there are r values per vector. In our implementation, $r = 16$, as we use FP32 data stored in 512-bit vector registers. Note that we assume that there are $\hat{O}_C$ vector registers, $\mathtt{v}_{\mathtt{out}}^{(0)} \cdots \mathtt{v}_{\mathtt{out}}^{(\hat{\mathtt{O}}_\mathtt{C})}$, available to hold intermediate results. Otherwise, we can subdivide the output tensor `O` along its channel dimension into blocks small enough to be held in register, and execute Algorithm 2 independently for each block.

SIMD Compression. Because we must compress our input data at runtime, we also require an efficient algorithm to compress a matrix M to CSR format. This can be done as follows: given a SIMD vector, v, of 16 floats, which we want to compress, we use the **vcmp** instruction to identify the locations of the non-zero elements in v stored in a mask register m. Then use the **vcompress** instruction twice: once applied to v with mask m to produce contiguous non-zero elements to be written to `values`, and a second time applied to the vector $\{j, \ldots, j+15\}$ with mask m (where j is the column index of the first element of v in M) to produce column indices to be written to `columns`. The

popcnt instruction applied to m can be used to keep track of the number of non-zero elements and thereby maintain the offset for writing to `values` and `columns`, as well as to record $\texttt{row\_pointers}[i]$.

Algorithm 1 Sparse Convolution

```
1: for $(\texttt{ox}, \texttt{kx}) \in [0, O_x) \times [0, K_x)$ do
2:   $\texttt{ix} \leftarrow \texttt{ox} + \texttt{kx}$
3:   for $\texttt{in\_loc} \in [\texttt{row\_pointers}[\texttt{ix}], \texttt{row\_pointers}[\texttt{ix}+1])$ do
4:     $\texttt{ic} \leftarrow \texttt{columns}[\texttt{ix}][\ell]$
5:     for $\texttt{oc} \in [0, O_C)$ do
6:       $\texttt{O}[\texttt{oc}][\texttt{ox}] \mathrel{+}= \texttt{values}[\texttt{in\_loc}] * \texttt{K}[\texttt{oc}][\texttt{ic}][\texttt{kx}]$
7:     end for
8:   end for
9: end for
```

Algorithm 2 AVX Sparse Convolution

```
1: for $(\texttt{ox}, \texttt{kx}) \in [0, O_x) \times [0, K_x)$ do
2:   initialize $\texttt{v}_{\texttt{out}}^{(0)} \cdots \texttt{v}_{\texttt{out}}^{(\hat{\texttt{O}}_\texttt{C})}$ to 0
3:   $\texttt{ix} \leftarrow \texttt{ox} + \texttt{kx}$
4:   for $\texttt{in\_loc} \in [\texttt{row\_pointers}[\texttt{ix}], \texttt{row\_pointers}[\texttt{ix}+1])$ do
5:     $\texttt{v}_{\texttt{in}} \leftarrow \textbf{vbroadcast}(\texttt{values}[\texttt{in\_loc}])$
6:     $\texttt{ic} \leftarrow \texttt{columns}[\texttt{ix}][\ell]$
7:     for $\texttt{oc} = 1$ to $\hat{O}_C$ do
8:       // $\texttt{K}[\texttt{oc}][\texttt{ic}][\texttt{kx}]$ points to a kernel vector in memory
9:       $\texttt{v}_{\texttt{out}}^{\texttt{oc}} \leftarrow \textbf{vfmadd}(\texttt{v}_{\texttt{in}}, \texttt{K}[\texttt{oc}][\texttt{ic}][\texttt{kx}], \texttt{v}_{\texttt{out}}^{\texttt{oc}})$
10:    end for
11:  end for
12:  Store $\texttt{v}_{\texttt{out}}^{(0)} \cdots \texttt{v}_{\texttt{out}}^{(\hat{\texttt{O}}_\texttt{C})}$ to memory locations $\texttt{O}[0 \cdots \hat{O}_C][\texttt{ox}]$
13: end for
```

Next, we discuss a number of optimizations which we apply to our framework, focusing on CPU-based implementations.

Multicore. Our sparse convolution framework is embarrassingly parallel: we partition O into blocks $O_1, \ldots, O_n$ and assign blocks to n threads. Each thread fully computes its corresponding block of O. In order to avoid many threads having to load the same input data, we minimize the overlaps between pre-images of the blocks O_i. Observing that two elements of O with different channel coordinates, but which share the same spatial coordinate, have identical pre-images, we partition O *spatially* as much as possible, rather than partitioning along channels. In some cases, image sizes are too small to get spatial partitions with enough work to saturate threads, in which case we can choose to additionally partition along channels after all.

Input Pre-loading. We observe that the input broadcast on line 4 of Algorithm 2 has the potential to be high latency since it must read from memory. Fortunately, modern CPUs can hide the latency of such memory accesses via *pipelining* them, i.e. executing instructions which do not depend on the results of the load while waiting for the values from memory to become available. In order to take full advantage of this pipeline, we re-order the memory loads to be as early as possible, by issuing each broadcast instruction s loop iterations before it is actually needed, at the cost of requiring s additional registers to hold pending input values.

Hot kernels in cache. In some layers of some networks, convolutional kernels are so large that they do not fit in cache. For instance, the last several convolutions of Resnet50 are either $2048 \times 512 \times 1 \times 1$ or $512 \times 512 \times 3 \times 3$, which, at 4 bytes per (floating point) value, are 4MB and 9MB respectively, yet L2 cache sizes of Intel machines are commonly only 1MB. Keeping kernels in L2 is critical for performance since every iteration of the inner-most loop accesses a different kernel value (line 4). To ensure that kernels remain hot, we use a combination of two techniques.

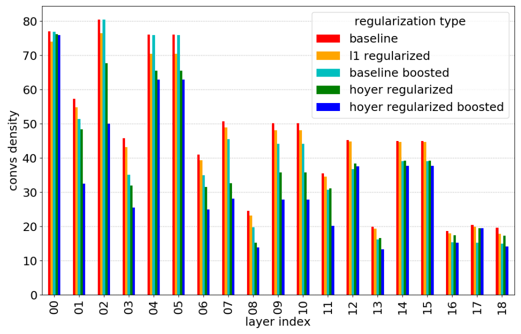
Firstly, if the kernel dimensions are such that the values associated with a single spatial pixel *do* fit in cache (i.e. $4I_C O_C < 1\text{MB}$), then we can order the outer loops of Algorithm 2 so that the loops over the spatial dimensions of the kernels is *outermost*. That is, for each of the K_x spatial coordinates of the kernel, we will compute partial outputs by performing all of the multiply-adds involving kernel values that share that coordinate, ultimately accumulating all of the partial results together. Thus, we only need to move the kernel values into L2 cache once and reuse them from there, at the cost of a few extra reads and writes of the (typically smaller) inputs and outputs.

Hot compression. To save on expensive memory accesses, we ensure that the results of the input pre-compression are used before being evicted from cache. To accomplish this, we subdivide the Sparse Convolutional operation into *subtasks*, each of which contains a block of data which fits entirely in cache. Then, we can process each block by first running the CSR compressor on only that block, and then immediately applying Algorithm 2 to the resulting compressed data while it is still hot.

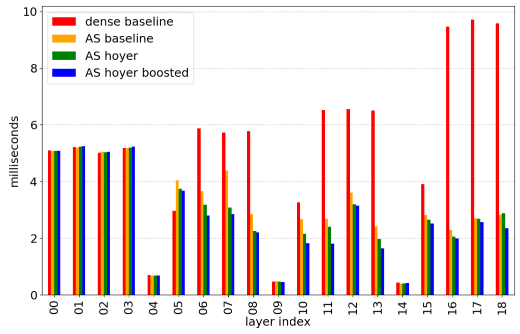
5. Experimental Results

Goals, Setup and Tasks. We experimentally validate our methods by applying them to a range of classic convolutional models for image classification. We aim to determine the extent to which our techniques can boost activation sparsity, and the impact this has in terms of layer-wise and end-to-end inference speedup on real models and tasks, compared against optimized baselines which do not leverage activation sparsity. We focus on the ResNet (He et al., 2016), and Mobilenet (Howard et al., 2017) architectures, applied to ImageNet ILSVRC2012 (Deng et al., 2009).

We implemented our thresholding methods in Pytorch, making use of the provided pre-trained models as starting points for the regularization and thresholding procedures. We implemented our sparse-input convolution in C++, on top of an existing fully-dense baseline framework, which uses optimized direct convolution or general matrix multiply (GeMM) operations for all layers. This framework gets an

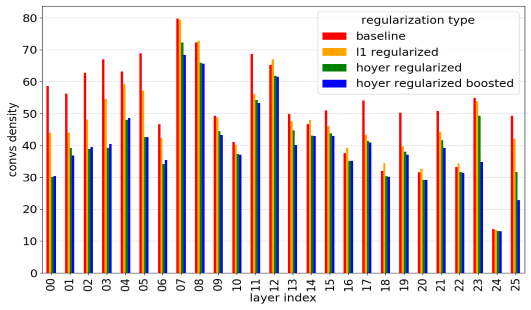


(a) Input Activation Map Sparsities for ResNet18/ImageNet.

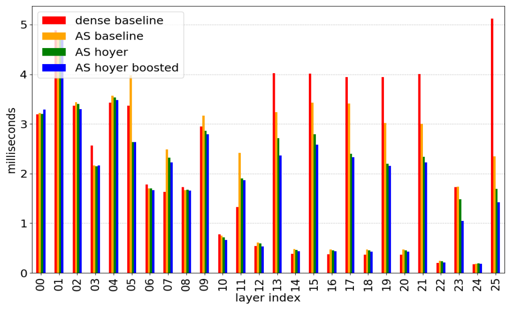


(b) Layer Latencies and Speedups for ResNet18/ImageNet.

Figure 4. Layerwise sparsities and speedups for ResNet18/ImageNet. The sparsified variant achieves significant speedups since it significantly reduces overhead in the more computationally-heavy layers.



(a) Activation Map Sparsities for Mobilenet/ImageNet.



(b) Layerwise Speedups for Mobilenet/ImageNet. Note: even-numbered layers are *depthwise* convolutions to which we do not apply our sparse algorithms.

Figure 5. A sample of our results for the Mobilenet model trained on the ImageNet dataset.

ONNX file (Bai et al., 2019) describing the network architecture as an input, parses and optimizes the graph, and then generates the Just-in Time compiled (JITted) assembly code for each layer. This baseline framework is well-optimized: as evident in Table 2, inference numbers using Dense match a state-of-the-art industrial solutions (MXNet 1.3 (Chen et al., 2015) using Intel MKL-DNN for CPU inference, and Pytorch/CUDA10 for GPU inference).

We perform our performance experiments on an AWS C5.12xlarge instance which sports an Intel Cascade Lake chip with 24 physical cores, has 96 GB of memory and runs Ubuntu 18.04, as well as on a local server with the same configuration. For GPU inference, we used P2.xlarge instance with one NVIDIA K80 GPU, running Pytorch 1.2.0 with CUDA10, using 16bit half precision.

Boosting Activation Sparsity. Our first experiments evaluate the ability of various methods to induce a large subset of activations to be zero. In particular, we study the average activation sparsity of 1) the baseline pre-trained models from Pytorch, 2) the L1-regularized models following the optimized hyperparameter values from (Georgiadis, 2019), 3) the (square) Hoyer-regularized models whose hyperparameters we identify through grid search, 4) the dynamically-boosted variants of the Hoyer-regularized models, following the algorithm from Section 3. For methods 2)–4) we performed fine-tuning for 20 epochs to recover or even increase accuracy under regularization. ((Georgiadis, 2019) recommends 90 epochs of retraining with regularization, but we were able to reproduce their results with this compressed

Model	Baseline	L1	Hoyer	Boosted Hoyer
ResNet18	53%	55%	62 %	**67 %**
ResNet50	53%	54%	61 %	**65 %**
Mobilenet	48%	52 %	58 %	**60 %**

Table 1. Average activation sparsities using different methods.

Model	MXNet+MKL-DNN	NVIDIA K80	Dense	Natural Sparsity	Hoyer Reg.	Boosted Hoyer
ResNet18	113.41	100.16	107.25	68.40	63.67	**60.92 (1.86x)**
ResNet50	317.49	350.2	256.06	194.86	183.21	**180.5 (1.75x)**
Mobilenet	88.55	114.3	62.64	58.93	51.80	**49.77 (1.78x)**

Table 2. Average inference running times in ms for batch size 64 on various models and variants (AWS C5.24xlarge for CPU and AWS P2.xlarge for GPU). Speedups are presented in brackets relative to the state-of-the art MXNet/MKL-DNN CPU inference framework.

schedule.) We present average values, with the note that results are extremely stable across sample batches (standard deviation $< 1\%$). For all of the models presented, the accuracy loss relative to the Torchvision baseline is $< 0.3\%$.

Table 1 presents average results for each technique, while Table 3 presents baseline and Boosted Hoyer results for wide and deep models.

A sample of layer-wise results are presented in Figures 4 (ResNet18) and 5 (Mobilenet) , while the average sparsities are presented in Table 1. One salient trend is that Hoyer and Dynamic Boosting are able to consistently boost sparsities, significantly beyond the baseline or L1 regularization. For instance, for the input layer of Mobilenet, they both reduce density by $\sim 2\times$ versus the natural sparsity, and by 50% versus L1 regularization. We note that, across all layers of all networks, there are only *two* layers where L1 regularization provides higher sparsity (the input layers of the residual networks), and by a very narrow margin. The second noticeable trend is that Dynamic Boosting can consistently reduce the density of activations without accuracy loss: for Mobilenet, these margins are almost negligible, but they become significant for the residual models, where boosting almost doubles the sparsity improvement of the best regularizer (Hoyer). A third observation (Table 3) is that our methods are especially effective in the context of accurate but heavy wide and deep models, where activation density can be effectively *halved* through boosting, without accuracy loss.

Model	Natural AS	Boosted	Speedup
ResNet50	53%	**65%**	**1.67x**
2x Wide ResNet50	58%	**81%**	**2.04x**
ResNet101	57%	**79%**	**1.53x**
2x Wide ResNet101	63%	**84%**	**2.57x**

Table 3. Average activation sparsity and speedup.

End-to-End Inference Performance. We now turn our attention to how well the activation sparsity numbers we saw in the previous section translate to actual speedups in end-to-end inference on the respective models. Figures 4 and 5 presents execution times layer-by-layer, whereas Tables 2 and 3 presents average total execution times for the models at batch size 64 under various configurations and speedup. Table 3 presents average natural and boosted sparsities for deep and wide residual models. For these experiments, we found that the MXNet benchmark does not efficiently support the wide/deep models we consider; we therefore present speedups relative to our own dense implementation, which provides a more competitive baseline. All experiments are executed at 12 threads. (Trends for other batch sizes and thread counts are similar, and therefore omitted.)

Generally, we find that activation sparsity can lead to significant and consistent speedups across the layers, roughly proportional to the amount of activation sparsity. A significant fraction of the speedup can already be obtained on top of the pretrained models, by exploiting their natural sparsity. At the same time, regularization and boosting consistently provide additional speedups, in particular for the computationally-heavy but accurate wide/deep models. In fact, fortunately, the layers with the largest computational overhead have high input sparsity (especially with boosting).

The end-to-end results are summarized in the last column of each table. Experiments confirm that Hoyer with Dynamic Boosting consistently provides the highest speedups for ResNets and Mobilenet, in the range of 1.67x (ResNet50) to 2.57x (WideResNet101), relative to our optimized dense implementation.

6. Conclusions and Future Work

We have presented a framework for augmenting and leveraging activation sparsity in DNNs for computational speedups. Our framework leverages two new techniques: on the machine learning side, a set of regularization and thresholding tools to boost the average and peak activation sparsity of networks; on the technical side, an algorithm for efficiently performing convolutions on sparse inputs, along with its optimized implementation in C++. Our techniques are implemented in an extensible, modular framework, which could be leveraged by researchers wishing to extend our results for both creating models with higher activation sparsity, or faster algorithms for sparse convolutions. Our framework is particularly well-suited for speeding-up inference on accurate, but heavy, deep and wide models.

In future work, we plan to explore additional strategies for memory-bound layers, and investigate the impact of quantization on sparsity on computational speedups.

References

Agostinelli, F., Hoffman, M., Sadowski, P., and Baldi, P. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014.

Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N. E., and Moshovos, A. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

Alwani, M., Chen, H., Ferdman, M., and Milder, P. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 22. IEEE Press, 2016.

Bai, J., Lu, F., Zhang, K., et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, X. Escort: Efficient sparse convolutional neural networks on gpus. *arXiv preprint arXiv:1802.10280*, 2018.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Dong, X., Liu, L., Li, G., Li, J., Zhao, P., Wang, X., and Feng, X. Exploiting the input sparsity to accelerate deep neural networks: poster. In Hollingsworth, J. K. and Keidar, I. (eds.), *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pp. 401–402. ACM, 2019. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295713. URL https://doi.org/10.1145/3293883.3295713.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Georgiadis, G. Accelerating convolutional neural networks via activation map compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7085–7095, 2019.

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

Gudovskiy, D., Hodgkinson, A., and Rigazio, L. Dnn feature map compression using learned representation over gf (2). In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 0–0, 2018.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015.

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. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254. IEEE, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hoyer, P. O. Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research*, 5 (Nov):1457–1469, 2004.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Krizhevsky, A., Nair, V., and Hinton, G. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2736–2744, 2017.

Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

Mellempudi, N., Kundu, A., Mudigere, D., Das, D., Kaul, B., and Dubey, P. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.

Mishra, A. K., Nurvitadhi, E., Cook, J. J., and Marr, D. WRPN: wide reduced-precision networks. *CoRR*, abs/1709.01134, 2017. URL `http://arxiv.org/abs/1709.01134`.

Molchanov, D., Ashukha, A., and Vetrov, D. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2498–2507. JMLR. org, 2017.

Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W., and Dally, W. J. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40. IEEE, 2017.

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*, 2016a.

Park, J., Li, S. R., Wen, W., Li, H., Chen, Y., and Dubey, P. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *arXiv preprint arXiv:1608.01409*, 1(2), 2016b.

Polino, A., Pascanu, R., and Alistarh, D. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

Rhu, M., O'Connor, M., Chatterjee, N., Pool, J., Kwon, Y., and Keckler, S. W. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 78–91. IEEE, 2018.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., and Wang, Y. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pp. 167–188. Springer, 2014.

Wu, X., Wu, Y., and Zhao, Y. High performance binarized neural networks trained on the imagenet classification task. *CoRR*, abs/1604.03058, 2016. URL `http://arxiv.org/abs/1604.03058`.

Yang, H., Wen, W., and Li, H. Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures. *arXiv preprint arXiv:1908.09979*, 2019.

Zagoruyko, S. and Komodakis, N. Wide Residual Networks. *ArXiv e-prints*, May 2016.

Zhang, H., Li, J., Kara, K., Alistarh, D., Liu, J., and Zhang, C. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*, pp. 4035–4043, 2017.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016. URL `http://arxiv.org/abs/1612.01064`.

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.