Lossless Compression of Deep Neural Networks

Thiago Serra[1], Abhinav Kumar[2], and Srikumar Ramalingam[2]

[1] Bucknell University, USA
thiago.serra@bucknell.edu
[2] The University of Utah, USA
abhinav.kumar@utah.edu,srikumar@cs.utah.edu

Abstract. Deep neural networks have been successful in many predictive modeling tasks, such as image and language recognition, where large neural networks are often used to obtain good accuracy. Consequently, it is challenging to deploy these networks under limited computational resources, such as in mobile devices. In this work, we introduce an algorithm that removes units and layers of a neural network while not changing the output that is produced, which thus implies a lossless compression. This algorithm, which we denote as LEO (Lossless Expressiveness Optimization), relies on Mixed-Integer Linear Programming (MILP) to identify Rectifier Linear Units (ReLUs) with linear behavior over the input domain. By using ℓ_1 regularization to induce such behavior, we can benefit from training over a larger architecture than we would later use in the environment where the trained neural network is deployed.

Keywords: Deep Learning · Mixed-Integer Linear Programming · Neural Network Pruning · Neuron Stability · Rectifier Linear Unit.

Table 1. Compression of 2-hidden-layer rectifier networks trained on MNIST. Each line summarizes tests on 31 networks. Depending on how the network is trained, the higher incidence of stable units allows for more compression while preserving the trained network accuracy. For example, training with ℓ_1 regularization induces such stability and then inactive units can be removed. Interestingly, the small amount of regularization that improves accuracy during training also helps compressing the network later.

Layer width	ℓ_1 weight	Accuracy (%)	Units removed		Network
			1st layer	2nd layer	compression (%)
25	0.001	95.76 ± 0.05	5.7 ± 0.3	5.1 ± 0.3	22 ± 1
25	0.0002	97.24 ± 0.02	1.2 ± 0.1	3.0 ± 0.4	8.3 ± 0.7
25	0	96.68 ± 0.03	0 ± 0	0 ± 0	0 ± 0
50	0.001	96.05 ± 0.04	16.9 ± 0.6	12.5 ± 0.6	29.4 ± 0.7
50	0.0002	97.81 ± 0.02	7.6 ± 0.4	7.5 ± 0.5	15.1 ± 0.6
50	0	97.62 ± 0.02	0 ± 0	0 ± 0	0 ± 0
100	0.0005	97.14 ± 0.02	36.7 ± 0.7	24.9 ± 0.6	30.8 ± 0.5
100	0.0001	98.14 ± 0.01	18.6 ± 0.5	11.1 ± 0.7	14.9 ± 0.4
100	0	98.00 ± 0.01	0 ± 0	0 ± 0	0 ± 0

1   Introduction

Deep Neural Networks (DNNs) have achieved unprecedented success in many domains of predictive modeling, such as computer vision [60, 17, 36, 91, 43], natural language processing [90], and speech [45]. While complex architectures are usually behind such feats, it is not fully known if these results depend on such DNNs being as wide or as deep as they currently are for some applications.

In this paper, we are interested in the compression of DNNs, especially to reduce their size and depth. More generally, that relates to the following question of wide interest about neural networks: given a neural network $\mathcal{N}_1$, can we find an *equivalent* neural network $\mathcal{N}_2$ with a different architecture? Since a trained DNN corresponds to a function mapping its inputs to outputs, we can formalize the equivalence among neural networks as follows [78]:

Definition 1 (Equivalence). *Two deep neural networks $\mathcal{N}_1$ and $\mathcal{N}_2$ with associated functions $\boldsymbol{f}_1 : \mathbb{R}^{n_0} \to \mathbb{R}^m$ and $\boldsymbol{f}_2 : \mathbb{R}^{n_0} \to \mathbb{R}^m$, respectively, are equivalent if $\boldsymbol{f}_1(\boldsymbol{x}) = \boldsymbol{f}_2(\boldsymbol{x}) \ \forall \ \boldsymbol{x} \in \mathbb{R}^{n_0}$.*

In other words, our goal is to start from a trained neural network and identify neural networks with fewer layers or smaller layer widths that would produce the exact same outputs. Since the typical input for certain applications is bounded along each dimension, such as $\boldsymbol{x} \in [0, 1]^{n_0}$ for the MNIST dataset [63], we can consider a broader family of neural networks that would be regarded as equivalent in practice. We formalize that idea with the concept of local equivalence:

Definition 2 (Local Equivalence). *Two deep neural networks $\mathcal{N}_1$ and $\mathcal{N}_2$ with associated functions $\boldsymbol{f}_1 : \mathbb{R}^{n_0} \to \mathbb{R}^m$ and $\boldsymbol{f}_2 : \mathbb{R}^{n_0} \to \mathbb{R}^m$, respectively, are local equivalent with respect to a domain $\mathbb{D} \subseteq \mathbb{R}^{n_0}$ if $\boldsymbol{f}_1(\boldsymbol{x}) = \boldsymbol{f}_2(\boldsymbol{x}) \ \forall \ \boldsymbol{x} \in \mathbb{D}$.*

For a given application, local equivalence with respect to the domain of possible inputs suffices to guarantee that two networks have the same accuracy in any test. Hence, finding a smaller network that is local equivalent to the original network implies a compression in which there is no loss. In this paper, we show that simple operations such as removing or merging units and folding layers of a DNN can yield such lossless compression under certain conditions. We denote as *folding* the removal of a layer by directly connecting the adjacent layers, which is accompanied by adjusting the weights and biases of those layers accordingly.

2   Background

We study feedforward DNNs with Rectifier Linear Unit (ReLU) activations [38], which are comparatively simpler than other types of activations. Nevertheless, ReLUs are currently the type of unit that is most commonly used [64], which is in part due to landmark results showing their competitive performance [77, 35].

Every network has input $\boldsymbol{x} = [x_1 \ x_2 \ \ldots \ x_{n_0}]^T$ from a bounded domain $\mathbb{X}$ and corresponding output $\boldsymbol{y} = [y_1 \ y_2 \ \ldots \ y_m]^T$, and each hidden layer $l \in \mathbb{L} = \{1, 2, \ldots, L\}$ has output $\boldsymbol{h}^l = [h_1^l \ h_2^l \ldots h_{n_l}^l]^T$ from ReLUs indexed by $i \in \mathbb{N}_l =$

$\{1, 2, \ldots, n_l\}$. Let $\boldsymbol{W}^l$ be the $n_l \times n_{l-1}$ matrix where each row corresponds to the weights of a neuron of layer l, and let $\boldsymbol{b}^l$ be vector of biases associated with the units in layer l. With $\boldsymbol{h}^0$ for $\boldsymbol{x}$ and $\boldsymbol{h}^{L+1}$ for $\boldsymbol{y}$, the output of each unit i in layer l consists of an affine function $g_i^l = \boldsymbol{W}_i^l \boldsymbol{h}^{l-1} + \boldsymbol{b}_i^l$ followed by the ReLU activation $h_i^l = \max\{0, g_i^l\}$. The unit i in layer l is denoted *active* when $\boldsymbol{h}_i^l > 0$ and *inactive* otherwise. DNNs consisting solely of ReLUs are denoted *rectifier* networks, and their associated functions are always piecewise linear [7].

2.1 Mixed-Integer Linear Programming

Our work is primarily based on the fast growing literature on applications of Mixed-Integer Linear Programming (MILP) to deep learning. MILP can be used to map inputs to outputs of each ReLU and consequently of rectifier networks. Such formulations have been used to produce the image [71, 27] and estimate the number of pieces [87, 86] of the piecewise linear function associated with the network, generate adversarial perturbations to test the network robustness [16, 31, 96, 89, 106, 6], and implement controllers based on DNN models [85, 109].

For each unit i in layer l, we can map $\boldsymbol{h}^{l-1}$ to g_i^l and h_i^l with a formulation that also includes a binary variable z_i denoting if the unit is active or not, a variable $\bar{h}_i^l$ denoting the output of a complementary fictitious unit $\bar{h}_i^l = max\left\{0, -g_i^l\right\}$, and constants H_i^l and $\bar{H}_i^l$ that are positive and as large as h_i^l and $\bar{h}_i^l$ can be:

$$\boldsymbol{W}_i^l \boldsymbol{h}^{l-1} + b_i^l = g_i^l \tag{1}$$

$$g_i^l = h_i^l - \bar{h}_i^l \tag{2}$$

$$h_i^l \le H_i^l z_i^l \tag{3}$$

$$\bar{h}_i^l \le \bar{H}_i^l (1 - z_i^l) \tag{4}$$

$$h_i^l \ge 0 \tag{5}$$

$$\bar{h}_i^l \ge 0 \tag{6}$$

$$z_i^l \in \{0, 1\} \tag{7}$$

This formulation can be strengthened by using the smallest possible values for H_i^l and $\bar{H}_i^l$ [31, 96] and valid inequalities to avoid fractional values of z_i^l [6, 86].

The largest possible value of g_i^l, which we denote $\mathcal{G}_i^l$, can be obtained as

$$\mathcal{G}_i^{l'} = \max \quad \boldsymbol{W}_i^{l'} \boldsymbol{h}^{l'-1} + b_i^{l'} \tag{8}$$

$$\text{s.t.} \quad (1)\text{--}(7) \quad \forall l \in \{1, \ldots, l'-1\}, \ i \in \mathbb{N}_l \tag{9}$$

$$x \in \mathbb{X} \tag{10}$$

If $\mathcal{G}_i^l > 0$, then $\mathcal{G}_i^l$ is also the largest value of h_i^l and it can be used for constant H_i^l. Otherwise, $h_i^l = 0$ for any input $x \in \mathbb{X}$. We can also minimize $\boldsymbol{W}_i^l \boldsymbol{h}^{l-1} + b_i^l$ to obtain $\overline{\mathcal{G}}_i^l$, the smallest possible value of g_i^l, and use $-\overline{\mathcal{G}}_i^l$ for constant $\bar{H}_i^l$ if $\overline{\mathcal{G}}_i^l < 0$; whereas $\overline{\mathcal{G}}_i^l > 0$ implies that $h_i^l > 0$ for any input $x \in \mathbb{X}$. By solving these formulations from the first to the last layer, we have the tightest values

for H_i^l and $\bar{H}_i^l$ for $l \in \{1, \dots l' - 1\}$ when we reach layer l'. Units with only zero or positive outputs were first identified using MILP in [96], where they are denoted as *stably inactive* and *stably active*, and their incidence was induced with ℓ_1 regularization. That was later exploited to accelerate the verification of robustness by making the corresponding MILP formulation easier to solve [106].

In this paper, we use the stability of units to either remove or merge them while preserving the outputs produced by the DNN. The same idea could be extended to other architectures with MILP mappings, such as Binarized Neural Networks (BNNs) [18, 78]. MILP has been used in BNNs for adversarial testing [56] and along with Constraint Programming (CP) for training [52]. BNNs have characteristics that also make them suitable under limited resources [78].

2.2   Related Work

Our work relates to the literature on neural network compression, and more specifically to methods that simplify a trained DNN. Such literature includes low-rank decomposition [22, 53, 112, 101, 81, 26], quantization [83, 18, 105, 97], architecture design [91, 51, 48, 49, 94], non-structured pruning [66], structured pruning [40, 65, 74, 39, 72, 1, 110], sparse learning [68, 114, 4, 102], automatic discarding of layers in ResNets [98, 111, 44], variational methods [113], and the recent Lottery Ticket Hypothesis [32] by which training only certain subnetworks in the DNN — the *lottery tickets* — might be good enough. However, network compression is often achieved with side effects to the function associated with the DNN.

In contrast to many lossy pruning methods that typically focus on removing unimportant neurons and connections, our approach focuses on developing lossless transformations that exactly preserve the expressiveness during the compression. A necessary criterion for equivalent transformation is that the resulting network is as expressive as the original one. Methods to study neural network expressiveness include universal approximation theory [19], VC dimension [9], trajectory length [82], and linear regions [79, 76, 75, 82, 7, 87, 41, 42, 86].

We can also consider our approach as a form of reparameterization, or equivalent transformation, in graphical models [59, 100, 103]. If two parameter vectors θ and θ' define the same energy function (i.e., $E(\boldsymbol{x}|\theta)) = E(\boldsymbol{x}|\theta'), \forall\, \boldsymbol{x}$), then θ' is called a reparameterization of θ. Reparameterization has played a key role in several inference problems such as belief propagation [100], tree-weighted message passing [99], and graph cuts [57]. The idea is also associated with characterizing the functions that can be represented by DNNs [46, 19, 95, 67, 7, 73, 62].

Finally, our work complements the vast literature at the intersection of mathematical optimization and Machine Learning (ML). General-purpose methods have been applied to train ML models to optimality [12, 13, 52, 84]. Conversely, ML models have been extensively applied in optimization [11, 34]. To mention a few lines of work, ML has been used to find feasible solutions [33, 24] and predict good solutions [25, 21]; determine how to branch on [55, 3, 69, 8, 47] or add cuts [93], when to linearize [14], or when to decompose [61] an optimization problem; how to adapt algorithms for each problem [54, 50, 88, 58, 20, 10, 23]; obtain better optimization bounds [15]; embed the structure of the problem as a layer

of a neural network [5, 108, 2, 29]; and predict the resolution by a time-limit [30], the feasibility of the problem [107], and the problem itself [28, 70, 92].

3   `LEO`: Lossless Expressiveness Optimization

Algorithm 1, which we denote `LEO` (Lossless Expressiveness Optimization), loops over the layers to remove units with constant outputs regardless of the input, some of the stable units, and any layers with constant output due to those two types of units. These modifications of the network architecture are followed by changes to the weights and biases of the remaining units in the network to preserve the outputs produced. The term *expressiveness* is commonly used to refer to the ability of a network architecture to represent complex functions [86].

First, `LEO` checks the weights and stability of each unit and decides whether to immediately remove them. A unit with constant output, which is either stably inactive or has zero input weights, is removed as long as there are other units left in the layer. A stably active unit with varying output is removed if the column of weights of that unit is linearly dependent on the column of weights of stably active units with varying outputs that have been previously inspected in that layer. We consider the removal of such stably active units as a *merging* operation since the output weights of other stable units need to be adjusted as well.

Second, `LEO` checks if layers can be removed in case the units left in the layer are all stable or have constant output. If they are all stably active with varying output, then the layer is removed by directly joining the layers before and after it, which we can regard as a *folding* operation. In the particular case in which only one unit is left with constant output, be it stably inactive or not, then all hidden layers are removed because the network has an associated function that is constant in $\mathbb{D}$. We denote the latter operation as *collapsing* the neural network.

Figure 1 shows examples of units being removed and merged on the left as well as of a layer being folded on the right. Although possible, folding or collapsing a trained neural network is not something that we would expect to achieve in practice unless we are compressing with respect to a very small domain $\mathbb{D} \subset \mathbb{X}$.

Theorem 1. *For a neural network $\mathcal{N}_1$, Algorithm 1 produces a neural network $\mathcal{N}_2$ such that $\mathcal{N}_1$ and $\mathcal{N}_2$ are local equivalent with respect to an input domain $\mathbb{D}$.*

Proof. If $\mathcal{G}_i^l < 0$, then $h_i^l = 0$ for any input in $\mathbb{D}$ and unit i in layer l can be regarded as stably inactive. Otherwise, if $\boldsymbol{W}_i^l = \boldsymbol{0}$, then the output of the unit is positive but constant. Those two types of units are analyzed by the block starting at line 5. If there are other units left in the layer, which are either not stable or stably active but not removed, then removing unit a stably inactive unit i does not affect the output of subsequent units since the output of the unit is always 0 in $\mathbb{D}$. Likewise, in the case that $\boldsymbol{W}_i^l = \boldsymbol{0}$ and $b_i^l > 0$, then the output of the network remains the same after removing that unit if such removal of h_i^l from each unit j in layer $l + 1$ is followed by adding $w_{ji}^{l+1} b_i^l$ to b_j^{l+1}.

If $\bar{\mathcal{G}}_i^l > 0$, then $h_i^l = \boldsymbol{W}_i^l h^{l-1} + b_i^l$ for any input in $\mathbb{D}$ and unit i in layer l can be regarded as stably active. Those units are analyzed by the block starting
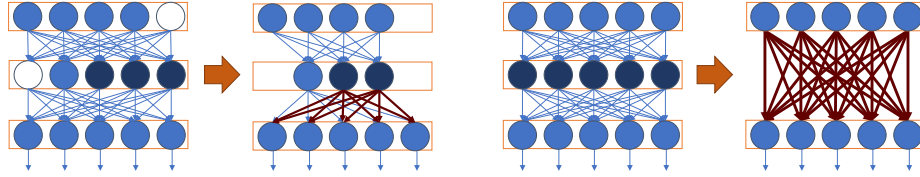
Fig. 1. Examples of output-preserving neural network compression obtained with LEO. On the left, two units in white are stably inactive and three units indexed by set S in darker blue are stably active, where $\text{rank}(\boldsymbol{W}_S^2)=2$. In such a case, we can remove the stably inactive units and merge the stably active units to produce the same input to the next layer using only two units. On the right, an entire layer is stably active. In such a case, we can fold the layer by directly connecting the layers before and after it. In both cases, the red arcs correspond to coefficients that need to be adjusted accordingly.

at line 14. If the rank of the submatrix $\boldsymbol{W}_S^l$ consisting of the weights of stably active units in set S is the same as that of $\boldsymbol{W}_{S\cup\{i\}}^l$ and given that $\boldsymbol{W}_i^l \neq \boldsymbol{0}$ for every $i \in S$, then $S \neq \emptyset$ and $h_i^l = \sum\limits_{k\in S} \alpha_k w_k^l h^{l-1} + b_i^l = \sum\limits_{k\in S} \alpha_k(h_k^l - b_k^l) + b_i^l$. Since there would be other units left in the layer, the output of the network remains the same after removing the unit if such removal of h_i^l from each unit j in layer $l+1$ is followed by adding $\alpha_k w_{ji}^{l+1}$ to w_{jk}^{l+1} and $w_{ji}^{l+1}\left(b_i^l - \sum\limits_{k\in A} \alpha_k b_k^l\right)$ to b_j^{l+1}.

If all units left in layer l are stably active and $|S| > 0$, then layer l is equivalent to an affine transformation and it is possible to directly connect layers $l-1$ and $l+1$, as in the block starting at line 30. Since $h_k^l = W_j^l h^{l-1} + b_k^l$ for each stably active unit k in layer l, then $h_i^{l+1} = W_i^{l+1} h^l + b_i^{l+1} = W_i^{l+1}\left(\sum\limits_{k=1}^{n_l} W_k^l h^{l-1} + b_k^l\right) + b_i^{l+1} = \sum\limits_{j\in n_{l-1}}\left(\sum\limits_{k\in S} w_{kj}^l w_{ik}^{l+1}\right) h_j^{l-1} + b_i^{l+1} + \left(\sum\limits_{k\in S} w_{ik}^{l+1} b_k^l\right)$.

If the only unit i left in layer l is stably inactive or stably active but has zero weights, then any input in $\mathbb{D}$ results in $h_i^l = \max\{0, b_i^l\}$, and consequently the neural network is associated with a constant function $f : x \to \Upsilon$ in $\mathbb{D}$. Therefore, it is possible to remove all hidden layers and replace the output layer with a constant function mapping to Υ as in the block starting at line 39. $\square$

Implementation We do not need to solve (8)–(10) to optimality to determine if $\mathcal{G}_i^l < 0$: it suffices to find a negative upper bound to guarantee that, or a solution with positive value to refute that. A similar reasoning applies to $\bar{\mathcal{G}}_i^l > 0$.

Algorithm 1 LEO produces a smaller equivalent neural network with respect to a domain $\mathbb{D}$ by removing units and layers while adjusting weights and biases

1: **for** $l \leftarrow 1, \ldots, L$ **do**
2:     $S \leftarrow \{\}$ $\triangleright$ Set of stable units left in layer l
3:     Unstable $\leftarrow$ False $\triangleright$ If there are unstable units in layer l
4:     **for** $i \leftarrow 1, \ldots, n_l$ **do**
5:         **if** $\mathcal{G}_i^l < 0$ for $x \in \mathbb{D}$ or $\boldsymbol{W}_i^l = \boldsymbol{0}$ **then** $\triangleright$ Unit i has constant output
6:             **if** $i < n_l$ **or** $|S| > 0$ **or** Unstable **then** $\triangleright$ Layer l still has other units
7:                 **if** $\boldsymbol{W}_i^l = \boldsymbol{0}$ and $b_i^l > 0$ **then**
8:                     **for** $j \leftarrow 1, \ldots, n_{l+1}$ **do** $\triangleright$ Adjust activations in layer $l + 1$
9:                         $b_j^{l+1} \leftarrow b_j^{l+1} + w_{ji}^{l+1} b_i^l$
10:                     **end for**
11:                 **end if**
12:                 Remove unit i from layer l $\triangleright$ Unit i is not necessary
13:             **end if**
14:         **else if** $\bar{\mathcal{G}}_i^l > 0$ for $x \in \mathbb{D}$ **then** $\triangleright$ Unit i is stably active
15:             **if** $\mathrm{rank}\big(\boldsymbol{W}_{S \cup \{i\}}^l\big) > |S|$ **then** $\triangleright$ w_i^l is linearly independent
16:                 $S \leftarrow S \cup \{i\}$ $\triangleright$ Keep unit in the network
17:             **else** $\triangleright$ Output of unit i is linearly dependent
18:                 Find $\{\alpha_k\}_{k \in S}$ such that $w_i^l = \sum_{k \in S} \alpha_k w_k^l$
19:                 **for** $j \leftarrow 1, \ldots, n_{l+1}$ **do** $\triangleright$ Adjust activations in layer $l + 1$
20:                     $w_{jk}^{l+1} \leftarrow w_{jk}^{l+1} + \sum_{k \in S} \alpha_k w_{ji}^{l+1}$
21:                     $b_j^{l+1} \leftarrow b_j^{l+1} + w_{ji}^{l+1}(b_i^l + \sum_{k \in S} \alpha_k b_k^l)$
22:                 **end for**
23:                 Remove unit i from layer l $\triangleright$ Unit i is no longer necessary
24:             **end if**
25:         **else**
26:             Unstable $\leftarrow$ True $\triangleright$ At least one unit is not stable
27:         **end if**
28:     **end for**
29:     **if not** Unstable **then** $\triangleright$ All units left in layer l are stable
30:         **if** $|S| > 0$ **then** $\triangleright$ The units left have varying outputs
31:             Create matrix $\bar{\boldsymbol{W}} \in \mathbb{R}^{n_{l-1} \times n_{l+1}}$ and vector $\bar{\boldsymbol{b}} \in \mathbb{R}^{n_{l+1}}$
32:             **for** $i \leftarrow 1, \ldots, n_{l+1}$ **do** $\triangleright$ Directly connect layers $l - 1$ and $l + 1$
33:                 $\bar{b}_i \leftarrow b_i^{l+1} + \sum_{k \in S} w_{ik}^{l+1} b_k^l$
34:                 **for** $j \leftarrow 1, \ldots, n_{l-1}$ **do**
35:                     $\bar{w}_{ij} \leftarrow \sum_{k \in S} w_{kj}^l w_{ik}^{l+1}$
36:                 **end for**
37:             **end for**
38:             Remove layer l; replace parameters in next layer with $\bar{\boldsymbol{W}}$ and $\bar{\boldsymbol{b}}$
39:         **else** $\triangleright$ Only unit left in layer l has constant output
40:             Compute output Υ for any input $\chi \in \mathbb{D}$ $\triangleright$ Function is constant
41:             $(\boldsymbol{W}^{L+1}, \boldsymbol{b}^{L+1}) \leftarrow (\boldsymbol{0}, \Upsilon)$ $\triangleright$ Set constant values in output layer
42:             Remove layers 1 to L and **break** $\triangleright$ Remove all hidden layers and leave
43:         **end if**
44:     **end if**
45: **end for**

4   Experiments

We conducted experiments to evaluate the potential for network compression using `LEO`. In these experiments, we trained rectifier networks on the MNIST dataset [63] with input size 784, two hidden layers of same width, and 10 outputs. The widths of the hidden layers are 25, 50, and 100. For each width, we identified in preliminary experiments a weight for ℓ_1 regularization on layer weights that improved the network accuracy in comparison with no regularization: 0.0002, 0.0002, and 0.0001, respectively. We trained 31 networks with that regularization weight, with 5 times the same weight to induce more stability, and with zero weight as a benchmark. We use the negative log-likelihood as the loss function after taking a softmax operation on the output layer, a batch size of 64 and SGD with a learning rate of 0.01, and momentum of 0.9 for training the model to 120 epochs. The learning rate is decayed by a factor of 0.1 after every 50 epochs. The weights of the network were initialized with the Kaiming initialization [43] and the biases were initialized to zero. The models were trained using Pytorch [80] on a machine with 40 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz processors and 132 GB of RAM. The MILPs were solved using Gurobi 8.1.1 [37] on a machine with Intel(R) Core(TM) i5-6200U CPU @ 2.30 GHz and 16 GB of RAM. We used callbacks to check bounds and solutions and then interrupt the solver after determining unit stability and bounds for each MILP.

Tables 1 and 2 summarize our experiments with mean and standard error. We note that the compression grows with the size of the network and the weight of ℓ_1 regularization, which induces the weights of each unit to be orders of magnitude smaller than its bias. The compression identified is all due to removing stably inactive units. Most of the runtime is due to solving MILPs for the second hidden layer. Given the incidence of stably active units, we conjecture that inducing rank deficiency in the weights or negative values in the biases could also be beneficial.

Table 2. Additional details about the experiments for each type of network, including runtime per test, incidence of stably active units, and overall network stability.

			Stably active units		Network
Layer width	ℓ_1 weight	Runtime (s)	1st layer	2nd layer	stability (%)
25	0.001	27.9 ± 0.3	2.5 ± 0.3	7.4 ± 0.4	41.3 ± 0.6
25	0.0002	29 ± 1	0 ± 0	1.0 ± 0.2	10.4 ± 0.7
25	0	28.4 ± 0.3	0 ± 0	0 ± 0	0 ± 0
50	0.001	103 ± 2	15 ± 0.5	24.9 ± 0.6	69.3 ± 0.4
50	0.0002	106 ± 3	2.7 ± 0.3	8.8 ± 0.5	26.6 ± 0.6
50	0	112 ± 3	0 ± 0	0 ± 0	0 ± 0
100	0.0005	421 ± 4	35.7 ± 0.6	57.7 ± 0.7	77.5 ± 0.2
100	0.0001	456 ± 8	11.1 ± 0.5	18 ± 0.7	29.4 ± 0.5
100	0	385 ± 2	0 ± 0	0 ± 0	0 ± 0

5   Conclusion

We introduced a lossless neural network compression algorithm, `LEO`, which relies on MILP to identify parts of the neural network that can be safely removed after reparameterization. We found that networks trained with ℓ_1 regularization are particularly amenable to such compression. In a sense, we could interpret ℓ_1 regularization as inducing a subnetwork to represent the function associated with the DNN. Future work may explore the connection between these subnetworks identified by `LEO` and lottery tickets, other forms of inducing posterior compressibility while training, and bounding techniques such as in [104] to efficiently identify stable units. Concomitantly, we have shown another form in which discrete optimization can play a key role in deep learning applications.

References

1. Aghasi, A., Abdi, A., Nguyen, N., Romberg, J.: Net-trim: Convex pruning of deep neural networks with performance guarantee. In: NeurIPS (2017)
2. Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., Kolter, Z.: Differentiable convex optimization layers. In: NeurIPS (2019)
3. Alvarez, A., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS Journal on Computing (2017)
4. Alvarez, J., Salzmann, M.: Learning the number of neurons in deep networks. In: NeurIPS (2016)
5. Amos, B., Kolter, Z.: Optnet: differentiable optimization as a layer in neural networks. In: ICML (2017)
6. Anderson, R., Huchette, J., Tjandraatmadja, C., Vielma, J.: Strong mixed-integer programming formulations for trained neural networks. In: IPCO (2019)
7. Arora, R., Basu, A., Mianjy, P., Mukherjee, A.: Understanding deep neural networks with rectified linear units. In: ICLR (2018)
8. Balcan, M.F., Dick, T., Sandholm, T., Vitercik, E.: Learning to branch. In: ICML (2018)
9. Bartlett, P., Maiorov, V., Meir, R.: Almost linear VC-dimension bounds for piecewise polynomial networks. Neural computation (1998)
10. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. In: ICLR (2017)
11. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. CoRR **abs/1811.06128** (2018)
12. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning (2017)
13. Bienstock, D., Muñoz, G., Pokutta, S.: Principled deep neural network training through linear programming. CoRR **abs/1810.03218** (2018)
14. Bonami, P., Lodi, A., Zarpellon, G.: Learning a classification of mixed-integer quadratic programming problems. In: CPAIOR (2018)
15. Cappart, Q., Goutierre, E., Bergman, D., Rousseau, L.M.: Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In: AAAI (2019)
16. Cheng, C., Nührenberg, G., Ruess, H.: Maximum resilience of artificial neural networks. In: ATVA (2017)
17. Ciresan, D., Meier, U., Masci, J., Schmidhuber, J.: Multi column deep neural network for traffic sign classification. Neural Networks (2012)

18. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. NeurIPS (2016)
19. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems (1989)
20. Dai, H., Khalil, E.B., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: NeurIPS (2017)
21. Demirović, E., Stuckey, P., Bailey, J., Chan, J., Leckie, C., Ramamohanarao, K., Guns, T.: An investigation into prediction + optimisation for the knapsack problem. In: CPAIOR (2019)
22. Denton, E., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. In: NeurIPS (2014)
23. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.M.: Learning heuristics for the TSP by policy gradient. In: CPAIOR (2018)
24. Ding, J.Y., Zhang, C., Shen, L., Li, S., Wang, B., Xu, Y., Song, L.: Accelerating primal solution findings for mixed integer programs based on solution prediction. CoRR **abs/1906.09575** (2019)
25. Donti, P., Amos, B., Kolter, Z.: Task-based end-to-end model learning in stochastic optimization. In: NeurIPS (2017)
26. Dubey, A., Chatterjee, M., Ahuja, N.: Coreset based neural network compression. In: ECCV (2018)
27. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward networks. In: NFM (2018)
28. Elmachtoub, A., Grigas, P.: Smart predict, then optimize. CoRR **abs/1710.08005** (2017)
29. Ferber, A., Wilder, B., Dilkina, B., Tambe, M.: MIPaaL: Mixed integer program as a layer. In: AAAI (2020)
30. Fischetti, M., Lodi, A., Zarpellon, G.: Learning MILP resolution outcomes before reaching time-limit. In: CPAIOR (2019)
31. Fischetti, M., Jo, J.: Deep neural networks and mixed integer linear optimization. Constraints (2018)
32. Frankle, J., Carbin, M.: The lottery ticket hypothesis: Finding sparse, trainable neural networks. In: ICLR (2019)
33. Galassi, A., Lombardi, M., Mello, P., Milano, M.: Model agnostic solution of CSPs via deep learning: A preliminary study. In: CPAIOR (2018)
34. Gambella, C., Ghaddar, B., Naoum-Sawaya, J.: Optimization models for machine learning: A survey. CoRR **abs/1901.05331** (2019)
35. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: AISTATS (2011)
36. Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., Bengio, Y.: Maxout networks. In: ICML (2013)
37. Gurobi Optimization, L.: Gurobi optimizer reference manual (2018), http://www.gurobi.com
38. Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., Seung, S.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature **405** (2000)
39. Han, S., Pool, J., Narang, S., Mao, H., Tang, S., Elsen, E., Catanzaro, B., Tran, J., Dally, W.: Dsd: regularizing deep neural networks with dense-sparse-dense training flow. arXiv preprint arXiv:1607.04381 (2016)

40. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: NeurIPS (2015)
41. Hanin, B., Rolnick, D.: Complexity of linear regions in deep networks. In: ICML (2019)
42. Hanin, B., Rolnick, D.: Deep relu networks have surprisingly few activation patterns. In: NeurIPS (2019)
43. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR (2016)
44. Herrmann, C., Bowen, R., Zabih, R.: Deep networks with probabilistic gates. CoRR **abs/1812.04180** (2018)
45. Hinton, G., Deng, L., Dahl, G., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition. IEEE Signal Processing Magazine (2012)
46. Hornik, K., Stinchcombe, M., White, H.: Multilayer feed-forward networks are universal approximators. Neural Networks (1989)
47. Hottung, A., Tanaka, S., Tierney, K.: Deep learning assisted heuristic tree search for the container pre-marshalling problem. Computers & Operations Research (2020)
48. Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
49. Huang, G., Liu, Z., Maaten, L.V.D., Weinberger, K.: Densely connected convolutional networks. In: CVPR (2017)
50. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LIOn (2011)
51. Iandola, F., Han, S., Moskewicz, M., Ashraf, K., Dally, W., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 MB model size. arXiv preprint arXiv:1602.07360 (2016)
52. Icarte, R., Illanes, L., Castro, M., Cire, A., McIlraith, S., Beck, C.: Training binarized neural networks using MIP and CP. In: International Conference on Principles and Practice of Constraint Programming (CP) (2019)
53. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. BMVC (2014)
54. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC — Instance-Specific Algorithm Configuration. In: ECAI (2010)
55. Khalil, E., Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: AAAI (2016)
56. Khalil, E., Gupta, A., Dilkina, B.: Combinatorial attacks on binarized neural networks. In: ICLR (2019)
57. Kolmogorov, V., Rother, C.: Minimizing nonsubmodular functions with graph cuts-a review. TPAMI (2007)
58. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine **35**(3) (2014)
59. Koval, V., Schlesinger, M.: Two-dimensional programming in image analysis problems. USSR Academy of Science, Automatics and Telemechanics (1976)
60. Krizhevsky, A., Sutskever, I., Hinton, G.: Imagenet classification with deep convolutional neural networks. In: NeurIPS (2012)
61. Kruber, M., Lübbecke, M., Parmentier, A.: Learning when to use a decomposition. In: CPAIOR (2017)
62. Kumar, A., Serra, T., Ramalingam, S.: Equivalent and approximate transformations of deep neural networks. arXiv preprint arXiv:1905.11428 (2019)

63. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE (1998)
64. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521** (2015)
65. Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H.: Pruning filters for efficient convnets. arXiv preprint arXiv:1608.08710 (2016)
66. Lin, C., Zhong, Z., Wei, W., Yan, J.: Synaptic strength for convolutional neural network. In: NeurIPS (2018)
67. Lin, H., Jegelka, S.: Resnet with one-neuron hidden layers is a universal approximator. In: NeurIPS (2018)
68. Liu, B., Wang, M., Foroosh, H., Tappen, M., Pensky, M.: Sparse convolutional neural networks. In: CVPR (2015)
69. Lodi, A., Zarpellon, G.: On learning and branching: a survey. Top **25**(2) (2017)
70. Lombardi, M., Milano, M.: Boosting combinatorial problem modeling with machine learning. In: IJCAI (2018)
71. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. CoRR **abs/1706.07351** (2017)
72. Luo, J.H., Wu, J., Lin, W.: Thinet: A filter level pruning method for deep neural network compression. In: ICCV (2017)
73. Mhaskar, H., Poggio, T.: Function approximation by deep networks. CoRR **abs/1905.12882** (2019)
74. Molchanov, P., Tyree, S., Karras, T., Aila, T., Kautz, J.: Pruning convolutional neural networks for resource efficient transfer learning. arXiv preprint arXiv:1611.06440 (2016)
75. Montúfar, G.: Notes on the number of linear regions of deep neural networks. In: SampTA (2017)
76. Montúfar, G., Pascanu, R., Cho, K., Bengio, Y.: On the number of linear regions of deep neural networks. In: NeurIPS (2014)
77. Nair, V., Hinton, G.: Rectified linear units improve restricted boltzmann machines. In: ICML (2010)
78. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: AAAI (2018)
79. Pascanu, R., Montúfar, G., Bengio, Y.: On the number of response regions of deep feedforward networks with piecewise linear activations. In: ICLR (2014)
80. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. NeurIPS Workshops (2017)
81. Peng, B., Tan, W., Li, Z., Zhang, S., Xie, D., Pu, S.: Extreme network compression via filter group approximation. In: ECCV (2018)
82. Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., Dickstein, J.: On the expressive power of deep neural networks. In: ICML (2017)
83. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. In: ECCV (2016)
84. Ryu, M., Chow, Y., Anderson, R., Tjandraatmadja, C., Boutilier, C.: Caql: Continuous action Q-learning. CoRR **abs/1909.12397** (2019)
85. Say, B., Wu, G., Zhou, Y.Q., Sanner, S.: Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programming. In: IJCAI (2017)
86. Serra, T., Ramalingam, S.: Empirical bounds on linear regions of deep rectifier networks. In: AAAI (2020)
87. Serra, T., Tjandraatmadja, C., Ramalingam, S.: Bounding and counting linear regions of deep neural networks. In: ICML (2018)

88. Serra, T.: On defining design patterns to generalize and leverage automated constraint solving (2012)
89. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Robustness certification with refinement. In: ICLR (2019)
90. Sutskever, I., Vinyals, O., Le, Q.: Sequence to sequence learning with neural networks. In: NeurIPS (2014)
91. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: CVPR (2015)
92. Tan, Y., Delong, A., Terekhov, D.: Deep inverse optimization. In: CPAIOR (2019)
93. Tang, Y., Agrawal, S., Faenza, Y.: Reinforcement learning for integer programming: Learning to cut. CoRR **abs/1906.04859** (2019)
94. Tang, Z., Peng, X., Li, K., Metaxas, D.: Towards efficient u-nets: A coupled and quantized approach. TPAMI (2019)
95. Telgarsky, M.: Benefits of depth in neural networks. In: COLT (2016)
96. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: ICLR (2019)
97. Tung, F., Mori, G.: Clip-q: Deep network compression learning by in-parallel pruning-quantization. In: CVPR (2018)
98. Veit, A., Belongie, S.: Convolutional networks with adaptive computation graphs. CoRR **abs/1711.11503** (2017)
99. Wainwright, M., Jaakkola, T., Willsky, A.: Map estimation via agreement on (hyper)trees: Message-passing and linear-programming approaches. IEEE Trans. Information Theory (2005)
100. Wainwright, M., Jaakkola, T., Willsky, A.: Tree consistency and bounds on the performance of the max-product algorithm and its generalizations. Statistics and Computing (2004)
101. Wang, W., Sun, Y., Eriksson, B., Wang, W., Aggarwal, V.: Wide compression: Tensor ring nets. In: CVPR (2018)
102. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. In: NeurIPS (2016)
103. Werner, T.: A linear programming approach to max-sum problem: A review. Technical Report CTU-CMP-2005-25, Center for Machine Perception (2005)
104. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examplesvia the convex outer adversarial polytope. In: ICML (2018)
105. Wu, J., Leng, C., Wang, Y., Hu, Q., Cheng, J.: Quantized convolutional neural networks for mobile devices. In: CVPR (2016)
106. Xiao, K., Tjeng, V., Shafiullah, N., Madry, A.: Training for faster adversarial robustness verification via inducing ReLU stability. ICLR (2019)
107. Xu, H., Koenig, S., Kumar, T.S.: Towards effective deep learning for constraint satisfaction problems. In: CP (2018)
108. Xue, Y., van Hoeve, W.J.: Embedding decision diagrams into generative adversarial networks. In: CPAIOR (2019)
109. Ye, Z., Say, B., Sanner, S.: Symbolic bucket elimination for piecewise continuous constrained optimization. In: CPAIOR (2018)
110. Yu, R., Li, A., Chen, C.F., Lai, J.H., Morariu, V., Han, X., Gao, M., Lin, C.Y., Davis, L.: Nisp: Pruning networks using neuron importance score propagation. In: CVPR (2018)
111. Yu, X., Yu, Z., Ramalingam, S.: Learning strict identity mappings in deep residual networks. In: CVPR (2018)
112. Zhang, X., Zou, J., Ming, X., He, K., Sun, J.: Efficient and accurate approximations of nonlinear convolutional networks. In: CVPR (2015)

113. Zhao, C., Ni, B., Zhang, J., Zhao, Q., Zhang, W., Tian, Q.: Variational convolutional neural network pruning. In: CVPR (2019)
114. Zhou, H., Alvarez, J., Porikli, F.: Less is more: Towards compact CNNs. In: ECCV (2016)