Scaling Up Exact Neural Network Compression by ReLU Stability

Thiago Serra [1] **Abhinav Kumar** [2] **Srikumar Ramalingam** [3]

Abstract

We can compress a neural network while exactly preserving its underlying functionality with respect to a given input domain if some of its neurons are stable. However, current approaches to determine the stability of neurons in networks with Rectified Linear Unit (ReLU) activations require solving or finding a good approximation to multiple discrete optimization problems. In this work, we introduce an algorithm based on solving a single optimization problem to identify all stable neurons. Our approach is on median 21 times faster than the state-of-art method, which allows us to explore exact compression on deeper (5×100) and wider (2×800) networks within minutes. For classifiers trained under an amount of ℓ_1 regularization that does not worsen accuracy, we can remove up to 40% of the connections.

1. Introduction

For the past decade, the computing requirements associated with state-of-art machine learning models have grown faster than typical hardware improvements (Amodei et al., 2018). Although those requirements are often associated with training neural networks, they are nevertheless aligned with the size of such neural networks. Therefore, it becomes challenging to deploy them in modest computational environments, such as in mobile devices.

Meanwhile, we have learned that the expressiveness of the models associated with neural networks—when measured in terms of their number of linear regions —grows polynomially on the number of neurons and occasionally exponentially on the network depth (Pascanu et al., 2014; Montúfar et al., 2014; Raghu et al., 2017; Serra et al., 2018; Hanin & Rolnick, 2019a;b). Hence, we may wonder if the pressing need for larger models could not be countered by such gains in model complexity. Namely, if we could not represent the same model using a smaller neural network. More specifically, we consider the following definition of equivalence:

Definition 1 (Serra et al. 2020)**.** *Two neural networks $\mathcal{N}_1$ and $\mathcal{N}_2$ with associated functions $\boldsymbol{f}_1 : \mathbb{R}^{n_0} \to \mathbb{R}^m$ and $\boldsymbol{f}_2 : \mathbb{R}^{n_0} \to \mathbb{R}^m$ are local equivalent with respect to a domain $\mathbb{D} \subseteq \mathbb{R}^{n_0}$ if $\boldsymbol{f}_1(x) = \boldsymbol{f}_2(x)\ \forall x \in \mathbb{D}$.*

There is an extensive literature on methods for compressing neural networks (Cheng et al., 2018; Blalock et al., 2020), which is aimed at obtaining smaller networks that are nearly as good as the original ones. These methods generally produce networks that not equivalent, and thus require retraining the neural network for better accuracy. They may also disproportionately affect some inputs more than others.

Compressing a neural network while preserving its associated function is a relatively less explored topic, which has been commonly referred to as *lossless compression* (Serra et al., 2020; Sourek & Zelezny, 2021). However, that term has also been used for the more general case in which the overall accuracy of the compressed network is no worse than that of the original network regardless of equivalence (Xing et al., 2020). Hence, we regard *exact compression* as a more appropriate term when equivalence is preserved.

Exact compression has distinct benefits and challenges. On the one hand, there is no need for retraining and no risk of disproportionately affecting some inputs more than others. On the other hand, optimization problems that are formulated for exact compression need to account for any valid input as opposed to relying on a sample of inputs. In this paper, we focus on how to scale such an approach to a point in which exact compression starts to become practical.

In particular, we introduce and evaluate a faster algorithm for exact compression based on identifying all neurons with Rectified Linear Unit (ReLU) activation (Hahnloser et al., 2000) that have a linear behavior, which are denoted as *stable* (Tjeng et al., 2019). In other words, those are the neurons for which the mapping of inputs to outputs is always characterized by a linear function, which is either the constant value 0 or the preactivation output. We can remove or merge such neurons—and even entire layers in some cases—while obtaining a smaller but equivalent neural network (Serra et al., 2020). Our main contributions are:

(i) We propose a single Mixed-Integer Linear Program-

[1]Bucknell University [2]Michigan State University [3]Google Research. Correspondence to: Thiago Serra <thiago.serra@bucknell.edu>.

ming (MILP) formulation to verify the stability of all neurons of a feedforward neural network with ReLU activations, which is faster than solving two MILP formulations per neuron—either optimally (Tjeng et al., 2019) or approximately (Serra et al., 2020). Compared to Serra et al. (2020), the median improvement is 12 times—and in fact greater in larger networks.

(ii) We further reduce the runtime by generating feasible solutions through the linear relaxation of the MILP formulation. The median improvement becomes 21.

(iii) We outline an algorithm that leverages the new MILP formulation to perform all compressions once per layer instead of once per stable neuron (Serra et al., 2020) and prove the correctness of its most elaborate steps.

(iv) We leverage the scalability of our approach to investigate exact compressibility on classifiers that are deeper (5×100) and wider (2×800) than previously studied in Serra et al. (2020) (2×100). We show that approximately 20% of the neurons and 40% of the connections can be removed when the network is trained with an amount of ℓ_1 regularization that does not worsen accuracy.

2. Related Work

There is an extensive literature on pruning methods for sparsifying or reducing the size of a neural network by removing connections, neurons, or even layers. These methods are justified by the significant amount of redundancy between network parameters (Denil et al., 2013) and the better generalization bounds of compressed networks (Arora et al., 2018; Zhou et al., 2019; Suzuki et al., 2020a;b).

Blalock et al. (2020) note that these methods are typically based on a tradeoff between model efficiency and quality: the models of compressed neural networks tend to have a comparatively lower accuracy, save some exceptions (Han et al., 2015; Xing et al., 2020; Suzuki et al., 2020a). Nevertheless, Hooker et al. (2019) observe that such compression leads to networks in which the loss in accuracy is disproportionately distributed across classes and more severe in a fraction of them; the most impacted inputs are those that the original network could not classify well; and the overall robustness to noise or adversarial examples is diminished.

In order to make up for model changes and potential accuracy loss, most approaches rely on a three-step procedure consisting of (1) training the neural network; (2) compression; and (3) retraining. Nevertheless, the scope of compression methods is seldom restricted to the second step. For example, the compressibility of a neural network hinges on how it was trained, with regularizations such as ℓ_1 and ℓ_2 often used to make part of the network parameters negligible in magnitude—and hopefully in impact as well.

In fact, the two main—and recurring—themes in this topic are pruning connections when the corresponding parameters are sufficiently small (Hanson & Pratt, 1988; Mozer & Smolensky, 1989; Janowsky, 1989; Han et al., 2015; 2016; Li et al., 2017; Frankle & Carbin, 2019; Gordon et al., 2020; Tanaka et al., 2020) and when the impact of the connection on the loss function is sufficiently small (LeCun et al., 1989; Hassibi et al., 1993; Lebedev & Lempitsky, 2016; Molchanov et al., 2017; Dong et al., 2017; Yu et al., 2018; Zeng & Urtasun, 2018; Lee et al., 2019; Wang et al., 2019; 2020). The main issue with the first approach is that small weights may nevertheless be important. Rosenfeld et al. (2020) studies the impact of their removal on the loss function. The main issue with the second approach is the computational cost of calculating the second-order derivatives of the loss function in deep networks, which has lead to many approaches to approximating the loss function.

Overlapping with such approximations, there is a growing literature on casting neural network compression as an optimization problem (He et al., 2017; Luo et al., 2017; Aghasi et al., 2017; Yu et al., 2018; Serra et al., 2020; ElAraby et al., 2020). In the majority of the cases, these formulations aim to minimize the impact of the compression on how the neural network performs on the training set.

Other lines of work and overlapping themes in neural network compression include combining similar neurons (Srinivas & Babu, 2015; Mariet & Sra, 2016; Suau et al., 2020; Suzuki et al., 2020a); low-rank approximations and random projections of the weight matrices (Jaderberg et al., 2014; Denton et al., 2014; Lebedev et al., 2015; Arora et al., 2018; Wang et al., 2019; Suzuki et al., 2020a;b); and statistical tests on the relevance of a connection to network output (Xing et al., 2020). Furthermore, there is a growing literature on pruning neural networks at initialization instead of after training (Lee et al., 2019; 2020; Wang et al., 2020; Tanaka et al., 2020; Frankle et al., 2021) as well as on what parameters to use when these networks are retrained (Frankle & Carbin, 2019; Liu et al., 2019b; Renda et al., 2020).

Although exact compression has only been recently explored for fully-connected feedforward neural networks (Serra et al., 2020) and graph neural networks (Sourek & Zelezny, 2021), we can regard it as an outgrowth of the literature on equivalent neural networks. That topic covers verifying that networks are equivalent (Narodytska et al., 2018; Büning et al., 2020), identifying operations that produce equivalent networks (Hecht-Nielsen, 1990; Chen et al., 1993; Kůrková & Kainen, 1994; Kumar et al., 2019; Phuong & Lampert, 2020), reconstructing networks from their outputs (Albertini & Sontag, 1993a;b; Fefferman & Markel, 1994; Al-Falou & Trummer, 2003; Rolnick & Kording, 2020), and evaluating the effect of redundant representations on network training (Berner et al., 2019; Petersen et al., 2020).

3. Setting and Notations

We consider fully-connected feedforward neural networks with L hidden layers, in which we denote n_l as the number of units—or width—of layer $l \in \mathbb{L} := \{1, 2, \dots, L\}$ and x_i^l as the output of the i-th unit of layer l for $i \in \{1, 2, \dots, n_l\}$. For uniformity, we denote $\boldsymbol{x}^0 \in \mathbb{R}^{n_0}$ as the network input. We denote the output of the i-th unit of layer l as $x_i^l = \sigma^l(y_i^l)$, where the pre-activation output $y_i^l := \boldsymbol{w}_i^l \cdot x^{l-1} + b_i^l$ is defined by the learned weights $\boldsymbol{w}_i^l \in \mathbb{R}^{n_{l-1}}$ and the bias $b_i^l \in \mathbb{R}$ of the unit as well as the activation function $\sigma^l : \mathbb{R} \to \mathbb{R}$ associated with layer l. We further assume that the activation function of every hidden layer l is $\sigma^l(u) = \max\{0, u\}$, the ReLU. The output layer may have a different structure, such as the softmax layer (Bridle, 1990), which is nevertheless irrelevant for our purpose of compressing the hidden layers.

For every layer $l \in \mathbb{L}$, let $\boldsymbol{W}^l = [\boldsymbol{w}_1^l \boldsymbol{w}_2^l \dots \boldsymbol{w}_{n_l}^l]^T$ be the matrix of weights, $\boldsymbol{W}_{\mathbb{S}}^l$ be a submatrix of $\boldsymbol{W}^l$ consisting of the rows in set $\mathbb{S}$, and $\boldsymbol{b}^l = [b_1^l b_2^l \dots b_{n_l}^l]^T$ be the vector of biases. Finally, let $\boldsymbol{I}_m(\mathbb{S})$ be an $m \times m$ diagonal matrix in which the i-th diagonal element is 1 if $i \in \mathbb{S}$ and 0 if $i \notin \mathbb{S}$.

4. Identifying Stability for Exact Compression

This section explains the concept of stability and describes how MILP has been used to identify stable neurons.

If the output of neuron i in layer l is always linear on its inputs, we say that the neuron is stable. This happens in two ways for the ReLU activation. When $x_i^l = 0$ for any valid input, which implies that $y_i^l \leq 0$, we say that the neuron is *stably inactive*. When $x_i^l = y_i^l$ for any valid input, which implies that $y_i^l \geq 0$, we say that the neuron is *stably active*.

The qualifier *valid* is essential since not every input may occur in practice. There are nonempty halfspaces on $\boldsymbol{x}^{l-1}$ that would make a neuron active or inactive if $\boldsymbol{w}_i^l \neq \boldsymbol{0}$, $\{\boldsymbol{x}^{l-1} : \boldsymbol{w}_i^l \cdot \boldsymbol{x}^{l-1} + b_i^l \leq 0\}$ and $\{\boldsymbol{x}^{l-1} : \boldsymbol{w}_i^l \cdot \boldsymbol{x}^{l-1} + b_i^l \geq 0\}$, but it is possible that only one of them contains valid inputs. For the first layer, we only need to account for the valid inputs to the neural network. For example, the domain of a network trained on the MNIST dataset is $\{\boldsymbol{x}^0 : \boldsymbol{x}^0 \in [0, 1]^{784}\}$ (LeCun et al., 1998). For the remaining hidden layers, we also need to account for the combinations of outputs that can be produced by the preceding layers given the valid inputs and the parameters of the network. Hence, it is no longer straightforward to determine stability.

We can determine if a neuron of a trained neural network is stable by solving optimization problems to maximize and minimize its preactivation output (Tjeng et al., 2019). The main decision variables in these problems are the inputs for which the preactivation output is maximized or minimized. Consequently, there is also a decision variable associated with the output of every neuron. In order to model the ReLU activation function, additional binary variables are used to express if the output of the unit is equal to zero or to its preactivation output. The inclusion of those binary variables lead to discrete optimization problems.

MILP formulation of a single neuron For every neuron i of layer l, we can map every input vector $\boldsymbol{x}^{l-1}$ to the corresponding output x_i^l through a set of linear constraints that also include a binary variable z_i^l denoting if the unit is active or not, a variable for the pre-activation output y_i^l, a variable $\chi_i^l := \max\{0, -y_i^l\}$ denoting the output of a complementary fictitious unit, and positive constants M_i^l and μ_i^l that are as large as x_i^l and χ_i^l can be:

$$\boldsymbol{w}_i^l \cdot \boldsymbol{x}^{l-1} + b_i^l = y_i^l \tag{1}$$
$$y_i^l = x_i^l - \chi_i^l \tag{2}$$
$$x_i^l \leq M_i^l z_i^l \tag{3}$$
$$\chi_i^l \leq \mu_i^l (1 - z_i^l) \tag{4}$$
$$x_i^l \geq 0 \tag{5}$$
$$\chi_i^l \geq 0 \tag{6}$$
$$z_i^l \in \{0, 1\} \tag{7}$$

Constraint (1) matches the layer input $\boldsymbol{x}^{l-1}$ with the neuron preactivation output y_i^l. We then use the binary variable z_i^l to match y_i^l with the neuron output with either x_i^l or 0. When $z_i^l = 1$, constraints (4) and (6) imply that $\chi_i^l = 0$, and thus $x_i^l = y_i^l$ due to constraint (2). That only happens if $y_i^l \geq 0$ due to constraint (5). When $z_i^l = 0$, constraints (3) and (5) imply that $x_i^l = 0$, and thus $\chi_i^l = -y_i^l$. That only happens if $y_i^l \leq 0$ due to constraint (6).

MILP formulations to determine neuron stability Let $\mathbb{X} \subset \mathbb{R}^{n_0}$ be the set of valid inputs for the neural network, which we may assume to be bounded in every direction. We can obtain the maximum and the minimum values for the preactivation output $y_i^{l'}$ of neuron i in layer l' by solving the following optimization problems (Tjeng et al., 2019):

$$\overline{\mathcal{Y}}_i^{l'} := \max \quad \boldsymbol{w}_i^{l'} \cdot \boldsymbol{x}^{l'-1} + b_i^{l'} \tag{8}$$
$$\text{s.t.} \quad \boldsymbol{x}^0 \in \mathbb{X} \tag{9}$$
$$(1) - (7)\ \forall l \in [l' - 1], i \in [n_l] \tag{10}$$

$$\underline{\mathcal{Y}}_i^{l'} := \min \quad \boldsymbol{w}_i^{l'} \cdot \boldsymbol{x}^{l'-1} + b_i^{l'} \tag{11}$$
$$\text{s.t.} \quad \boldsymbol{x}^0 \in \mathbb{X} \tag{12}$$
$$(1) - (7)\ \forall l \in [l' - 1], i \in [n_l] \tag{13}$$

When $\overline{\mathcal{Y}}_i^{l'} \leq 0$, then $x_i^{l'} = 0$ for every $x^0 \in \mathbb{X}$ and the neuron is stably inactive. When $\underline{\mathcal{Y}}_i^{l'} \geq 0$, then $x_i^{l'} = y_i^{l'}$ for every $x^0 \in \mathbb{X}$ and the neuron is stably active.

Variations of the formulations above have been proposed for diverse tasks over neural networks, such as verifying them (Cheng et al., 2017), embedding their model

into a broader decision-making problem (Say et al., 2017; Bergman et al., 2019; Delarue et al., 2020), and measuring their expressiveness (Serra et al., 2018). When stable units are identified, Tjeng et al. (2019) observe that other optimization problems over trained neural networks become easier to solve. For example, Xiao et al. (2019) use weight regularization to induce neuron stability in order to accelerate adversarial robustness verification. The properties of such MILP formulations and the methods to solve them more effectively have been extensively discussed (Fischetti & Jo, 2018; Anderson et al., 2019; Botoeva et al., 2020; Serra & Ramalingam, 2020; Anderson et al., 2020).

For the purpose of identifying stable neurons, however, it is not scalable to analyze large neural networks by solving two optimization problems per neuron (Tjeng et al., 2019)—or even by just approximately solving each of them to ensure that $\overline{\mathcal{Y}}_i^{l'} \leq 0$ or $\underline{\mathcal{Y}}_i^{l'} \geq 0$ (Serra et al., 2020).

5. A New Algorithm for Exact Compression

Based on observations discussed in what follows (I to III), we propose a new formulation to identify stable neurons (Section 5.1), means to generate feasible solutions to this formulation (Section 5.2), and a compression algorithm exploiting all stable neurons in each layer at once (Section 5.3).

5.1. A New MILP Formulation

Consider the two observations below and their implications:

I: The overlap between optimization problems Although previous approaches require solving many optimization problems, their formulations are all very similar: we maximize and minimize the same objective function for each neuron, the feasible set of the problems for each layer are the same, and they are contained in the feasible set of problems for the subsequent layers. Hence, a single formulation involving the entire neural network can potentially be used to solve all of those optimization problems.

II: Proving stability is harder than disproving it Certifying that a neuron is stable is considerably more complex than certifying that a neuron is *not* stable. For the former, we need to exhaustively show that all inputs lead to the neuron always being active or always being inactive, which is achieved by solving a pair of optimization problems above for every neuron. For the latter, we just need a pair of inputs to the neural network such that the neuron is active with one of them and inactive with the other. Hence, it would be easier to rule out which neurons are not stable, and then obtain the complementary set of stable neurons as a byproduct.

Therefore, we consider the problem of finding an input that serves as a certificate of a neuron not being stable to as many neurons of unknown classification as possible. For that purpose, we define a decision variable $p_i^l \in \{0,1\}$ to denote if an input activates neuron i in layer l. Likewise, we define a decision variable $q_i^l \in \{0,1\}$ to denote if an input does not activate neuron i in layer l. Furthermore, we restrict the scope of the problem to states that have not been previously observed by using $\mathbb{P}^l \subseteq \{1,\dots,n_l\}$ as the set of neurons in layer l for which there is no known input that activates the neuron. Likewise, we use $\mathbb{Q}^l \subseteq \{1,\dots,n_l\}$ as the set of neurons in layer l for which there is no known input that does not activate the neuron. For brevity, let $\boldsymbol{P} := (\mathbb{P}^1,\dots,\mathbb{P}^L)$ and $\boldsymbol{Q} := (\mathbb{Q}^1,\dots,\mathbb{Q}^L)$ characterize an instance of such optimization problem, which is formulated as follows:

$$\mathcal{C}(\boldsymbol{P},\boldsymbol{Q}) = \max \quad \sum_{l\in\mathbb{L}}\left(\sum_{i\in\mathbb{P}^l} p_i^l + \sum_{i\in\mathbb{Q}^l} q_i^l\right) \tag{14}$$

$$\text{s.t.} \quad \boldsymbol{x}^0 \in \mathbb{X} \tag{15}$$

$$(1)-(7)\ \forall l \in \mathbb{L}, i \in [n_l] \tag{16}$$

$$0 \leq p_i^l \leq z_i^l\ \forall l \in \mathbb{L}, i \in \mathbb{P}^l \tag{17}$$

$$0 \leq q_i^l \leq 1 - z_i^l\ \forall l \in \mathbb{L}, i \in \mathbb{Q}^l \tag{18}$$

$$p_i^l, q_i^l \in \{0,1\} \tag{19}$$

Note that constraint (19) is actually not necessary and can be removed. We refer to Appendix A for more details.

The formulation above is aimed to obtain an input for the neural network that maximizes the number of neurons with an activation state that has not been previously observed. The following results lead to an algorithm that uses the formulation to identify all stable neurons of a neural network.

Proposition 1. *If $\mathcal{C}(\boldsymbol{P},\boldsymbol{Q}) = 0$, then every neuron $i \in \mathbb{P}^l$ is stably inactive and every neuron $i \in \mathbb{Q}^l$ is stably active.*

Proof. Constraint (17) is the only upper bound on p_i^l besides constraint (19). Hence, if there is any solution $(\bar{x},\bar{z},\bar{p},\bar{q})$ of (16)–(19) in which $\bar{z}_i^l = 1$ for some $i \in \mathbb{P}_i^l, l \in \mathbb{L}$, then either $\bar{p}_i^l = 1$ or there is another solution $(\bar{\bar{x}},\bar{\bar{z}},\bar{\bar{p}},\bar{\bar{q}})$ in which $\bar{\bar{p}}_i^l = 1$ and all other variables have the same value.

Likewise, constraint (17) is the only upper bound on q_i^l besides constraint (19). Hence, if there is any solution $(\bar{x},\bar{z},\bar{p},\bar{q})$ of (16)–(19) in which $\bar{z}_i^l = 0$ for some $i \in \mathbb{P}_i^l, l \in \mathbb{L}$, then either $\bar{q}_i^l = 1$ or there is another solution $(\bar{\bar{x}},\bar{\bar{z}},\bar{\bar{p}},\bar{\bar{q}})$ in which $\bar{\bar{q}}_i^l = 1$ and all other variables have the same value.

If $\mathcal{C}(\boldsymbol{P},\boldsymbol{Q}) = 0$, then for every solution $(\bar{x},\bar{z},\bar{p},\bar{q})$ it follows that $\bar{p}_i^l = 0\ \forall i \in \mathbb{P}^l, l \in \mathbb{L}$ and $\bar{q}_i^l = 0\ \forall i \in \mathbb{Q}^l, l \in \mathbb{L}$, and consequently $\bar{z}_i^l = 0\ \forall i \in \mathbb{P}^l, l \in \mathbb{L}$ and $\bar{z}_i^l = 1\ \forall i \in \mathbb{Q}^l, l \in \mathbb{L}$. Thus, the neurons in $\mathbb{P}^l$ are always inactive and the neurons in $\mathbb{Q}^l$ are always active for any valid input. □

Corollary 2. *The stability of all neurons of a neural network can be determined by solving formulation (14)–(19) at most $N+1$ times, where $N := \sum_{l\in\mathbb{L}} n_l$.*

Proof. Let us initially consider a formulation in which $\mathbb{P}^l = \mathbb{Q}^l = \{1, \ldots, n_l\}\ \forall l \in \mathbb{L}$ and then respectively remove from those sets each neuron i for which $p_i^l = 1$ and $q_i^l = 1$ in any solution obtained. When the formulation is first solved, we remove each neuron from either $\mathbb{P}^l$ or $\mathbb{Q}^l$, and therefore N states remain unobserved. In subsequent steps, either (i) $\mathcal{C}(\boldsymbol{P}, \boldsymbol{Q}) > 0$ and the number of unobserved states decreases; or (ii) $\mathcal{C}(\boldsymbol{P}, \boldsymbol{Q}) = 0$, and thus any neuron $i \in \mathbb{P}^l$ is stably inactive and any neuron $i \in \mathbb{Q}^l$ is stably active. □

Hence, our approach entails solving at most $N + 1$ of such MILP formulations instead of exactly $2N$ as in prior work.

Those results imply that we can iteratively solve the new formulation as part of an algorithm to identify all stable neurons. In fact, we can determine the stability of the entire neural network with a single call to the MILP solver. Except for the last time that formulation (14)–(19) is solved, there is no need to solve it to optimality: any solution with a positive objective function value can be used to reduce the number of unobserved states. Hence, all that we need is a way to inspect every feasible solution obtained by the MILP solver and then remove the solutions in which either $p_i^l = 1$ or $q_i^l = 1$ for states that were already observed. Both of those needs can be addressed in fully-fledged MILP solvers by implementing a lazy constraint callback. We refer to Appendix B for more details. When we finally reach $\mathcal{C}(\boldsymbol{P}, \boldsymbol{Q}) = 0$, the correctness of the MILP solver serves as a certificate of the stability of those remaining neurons.

5.2. Inducing Feasible MILP Solutions

The runtime with a single solver call depends on the frequency with which feasible solutions are obtained. Although at most $N + 1$ optimal solutions would suffice if we were to make consecutive calls to the solver until $\mathcal{C}(\boldsymbol{P}, \boldsymbol{Q}) = 0$, we should not expect the same from the first $N + 1$ feasible solutions found by the MILP solver while using the lazy constraint callback because they may not have a positive objective function value given the p_i^l and q_i^l variables that have been fixed to 0. On top of that, obtaining a feasible solution for an MILP formulation is NP-complete (Cook, 1971). Hence, it is a good practice to use domain knowledge about the MILP formulation to produce feasible solutions for the MILP solver, such as the observation below:

III: Finding feasible solutions to MILP formulations on neural networks is easy As noted by Fischetti & Jo (2018), to any valid input of the neural network there is a corresponding solution of the MILP formulation: the neural network input implies which neurons are active and what is their output when active. Although any random input would suffice, we have found that it is better in practice to use inputs indirectly generated by the MILP solver. Namely, we can use the solution of the Linear Programming (LP) relaxation, which is solved at least once per branch-and-bound node. The LP relaxation is obtained from the MILP formulation by relaxing its integrality constraints. In the case of binary variables with domain $\{0, 1\}$, that consists of relaxing the domain of such variables to the continuous interval $[0, 1]$. We use the values of $\boldsymbol{x}^0$ in the solution of the LP relaxation as the network input, and thus obtain a feasible MILP solution by replacing the values of the other variables—which may be fractional for the decision variables with binary domains—by the values implied by fixing $\boldsymbol{x}^0$. However imprecise due to the relaxation of the binary domains, the input defined by the optimal solution of the LP relaxation may intuitively guide us toward maximizing the objective function on p_i^l and q_i^l.

5.3. Compressing the Neural Network

Algorithm 1 identifies all stable neurons of a neural network and then leverages that information for exact compression. The prior discussion on identifying stable units leads to the steps described between lines 2 and 18. First, $\boldsymbol{P}$ and $\boldsymbol{Q}$ are initialized between lines 2 and 4. Next, the MILP formulation is iteratively solved between lines 5 and 18. The block between lines 6 and 7 identifies the termination criterion, which implies that the unobserved states cannot be obtained with any valid input. The block between lines 8 and 14 inspects every feasible solution to identify unobserved states and then to effectively remove the decision variables associated with those states from the objective function by adding a constraint that sets their value to 0. The block between lines 15 and 16 produces a feasible solution from a solution of the LP relaxation when the latter is produced by the MILP solver. For brevity, we assume that the block between lines 8 and 14 would leverage such solution at the next repetition of the loop. Finally, the exact compression using the information about which neurons are stable is performed between lines 19 and 47. We describe next each form of compression contained in the algorithm. For ease of explanation, they are in reverse order of appearance.

These compression operations are the same as in Serra et al. (2020), but performed once per layer instead of once per neuron. In comparison to that work, the order of the operations is such that (i) neurons are not removed or merged if the entire layer is going to be folded; and (ii) special cases such as a neuron with weight vector $\boldsymbol{w}_i^l = \boldsymbol{0}$ do not need to be considered apart. For the most elaborate operations, we prove their correctness when applied to the entire layer.

Removal of stably inactive neurons This operation is performed in line 44. Since the output of stably inactive neurons is always 0, we remove those neurons without affecting subsequent computations. The case in which an entire layer is stably inactive is considered separately.

Algorithm 1 Identifies all stable neurons and subsequently performs exact compression of the neural network

1: **Input:** neural network $\left(L, \left\{(n_l, \boldsymbol{W}^l, \boldsymbol{b}^l)\right\}_{l\in\mathbb{L}}\right)$
2: **for** $l \leftarrow 1$ **to** L **do**
3: $\quad \mathbb{P}^l \leftarrow \mathbb{Q}^l = \{1, \ldots, n_l\}$
4: **end for**
5: **while** solving $\mathcal{C}(\boldsymbol{P}, \boldsymbol{Q})$ **do**
6: $\quad$ **if** optimal value is proven to be 0 **then**
7: $\quad\quad$ **break**
8: $\quad$ **else if** found positive MILP solution $(\bar{x}, \bar{z}, \bar{p}, \bar{q})$ **then**
9: $\quad\quad$ **for** $l \leftarrow 1$ **to** L **do**
10: $\quad\quad\quad \mathbb{P}^l \leftarrow \mathbb{P}^l \setminus \{i : i \in \mathbb{P}^l \text{ and } \bar{p}_i^l > 0\}$
11: $\quad\quad\quad$ set $p_i^l = 0$ for every $i \in \mathbb{P}^l$ such that $\bar{p}_i^l > 0$
12: $\quad\quad\quad \mathbb{Q}^l \leftarrow \mathbb{Q}^l \setminus \{i : i \in \mathbb{Q}^l \text{ and } \bar{q}_i^l > 0\}$
13: $\quad\quad\quad$ set $q_i^l = 0$ for every $i \in \mathbb{Q}^l$ such that $\bar{q}_i^l > 0$
14: $\quad\quad$ **end for**
15: $\quad$ **else if** found LP relaxation solution $(\tilde{x}, \tilde{z}, \tilde{p}, \tilde{q})$ **then**
16: $\quad\quad$ use $\tilde{x}^0$ to produce an MILP solution $(\bar{x}, \bar{z}, \bar{p}, \bar{q})$
17: $\quad$ **end if**
18: **end while**
19: **for** $l \leftarrow 1$ **to** L **do**
20: $\quad$ **if** $|\mathbb{P}^l| = n_l$ **then**
21: $\quad\quad$ find output $\overline{\boldsymbol{x}}^L$ for an arbitrary input $\overline{\boldsymbol{x}}^0 \in \mathbb{X}$
22: $\quad\quad$ remove all layers except L, which becomes 1
23: $\quad\quad \boldsymbol{W}^1 \leftarrow \boldsymbol{0}$
24: $\quad\quad \boldsymbol{b}^L \leftarrow \overline{\boldsymbol{x}}^L$
25: $\quad\quad$ **break**
26: $\quad$ **else if** $|\mathbb{P}^l| + |\mathbb{Q}^l| = n_l$ and $l < L$ **then**
27: $\quad\quad \boldsymbol{W}^{l+1} \leftarrow \boldsymbol{W}^{l+1}\boldsymbol{I}_{n_l}(\mathbb{Q}^l)\boldsymbol{W}^l$
28: $\quad\quad \boldsymbol{b}^{l+1} \leftarrow \boldsymbol{W}^{l+1}\boldsymbol{I}_{n_l}(\mathbb{Q}^l)\boldsymbol{b}^l + \boldsymbol{b}^{l+1}$
29: $\quad\quad$ remove layer l
30: $\quad$ **else if** $l < L$ **then**
31: $\quad\quad r \leftarrow \text{rank}\left(\boldsymbol{W}^l_{\mathbb{Q}^l}\right)$
32: $\quad\quad$ **if** $r < |\mathbb{Q}^l|$ and $l < L$ **then**
33: $\quad\quad\quad$ find $\overline{\mathbb{Q}} \subset \mathbb{Q}^l$ such that $r = |\overline{\mathbb{Q}}| = \text{rank}\left(\boldsymbol{W}^l_{\overline{\mathbb{Q}}}\right)$
34: $\quad\quad\quad$ **for every** $i \in \mathbb{Q}^l \setminus \overline{\mathbb{Q}}$ **do**
35: $\quad\quad\quad\quad$ find $\{\alpha_j^i\}_{j\in\overline{\mathbb{Q}}}$ such that $\boldsymbol{w}_i^l = \sum_{j\in\overline{\mathbb{Q}}} \alpha_j^i \boldsymbol{w}_j^l$
36: $\quad\quad\quad$ **end for**‘
37: $\quad\quad\quad$ **for** $k \leftarrow 1$ **to** n_{l+1} **do**
38: $\quad\quad\quad\quad$ **for every** $j \in \overline{\mathbb{Q}}$ **do**
39: $\quad\quad\quad\quad\quad w_{kj}^{l+1} \leftarrow w_{kj}^{l+1} + \sum_{i\in\mathbb{Q}^l\setminus\overline{\mathbb{Q}}} \alpha_j^i w_{ki}^{l+1}$
40: $\quad\quad\quad\quad$ **end for**
41: $\quad\quad\quad\quad b_k^{l+1} \leftarrow b_k^{l+1} + \sum_{i\in\mathbb{Q}^l\setminus\overline{\mathbb{Q}}} w_{ki}^{l+1}\left(b_i^l - \sum_{j\in\overline{\mathbb{Q}}} \alpha_j^i b_j^l\right)$
42: $\quad\quad\quad$ **end for**
43: $\quad\quad\quad$ remove from layer l every neuron $i \in \mathbb{Q}^l \setminus \overline{\mathbb{Q}}$
44: $\quad\quad\quad$ remove from layer l every neuron $i \in \mathbb{P}^l$
45: $\quad\quad$ **end if**
46: $\quad$ **end if**
47: **end for**

Merging of stably active neurons This operation is performed between lines 31 and 43. We use the following results to show how stably active neurons can be merged.

Proposition 3. *Let $\mathbb{S}$ be a set of stably active neurons in layer l. If $r := \text{rank}(\boldsymbol{W}_S^l) < |S|$ and let $\mathbb{T} \subset \mathbb{S}$ be a subset of those neurons for which $\text{rank}(\boldsymbol{W}_{\mathbb{T}}^l) = r$, then the output of the neurons in $\mathbb{S} \setminus \mathbb{T}$ is an affine function on the output of the neurons in $\mathbb{T}$.*

Proof. For every $i \in \mathbb{S} \setminus \mathbb{T}$, there is a vector $\boldsymbol{\alpha}^i \in \mathbb{R}^r$ such that $\boldsymbol{w}_i^l = \sum_{j\in\mathbb{T}} \alpha_j^i \boldsymbol{w}_j^l$. Since $\boldsymbol{x}_i^l = \boldsymbol{w}_i^l \cdot \boldsymbol{x}^{l-1} + b_i^l$ for every $i \in \mathbb{S}$ because all neurons in $\mathbb{S}$ are stably active, then for every $i \in \mathbb{S} \setminus \mathbb{T}$ it follows that $\boldsymbol{x}_i^l = \sum_{j\in\mathbb{T}} \alpha_j^i \boldsymbol{w}_j^l \cdot \boldsymbol{x}^{l-1} + b_i^l = \sum_{j\in\mathbb{T}} \alpha_j^i \left(\boldsymbol{w}_j^l \cdot \boldsymbol{x}^{l-1} + b_j^l\right) + \left(b_i^l - \sum_{j\in\mathbb{T}} \alpha_j^i b_j^l\right) = \sum_{j\in\mathbb{T}} \alpha_j^i x_j^l + \left(b_i^l - \sum_{j\in\mathbb{T}} \alpha_j^i b_j^l\right)$. □

Corollary 4. *If $\mathbb{S}$, $\mathbb{T}$, and l are such as in Proposition 3, then the pre-activation output of the neurons in layer $l+1$ is an affine function on the outputs of all neurons from layer l with exception of the neurons in $\mathbb{T}$.*

Proof. Let $\mathbb{U} := \{1, \ldots, n_l\} \setminus \mathbb{S}$. The pre-activation output of every neuron i in layer $l+1$ is given by $y_i^{l+1} = \sum_{j\in\mathbb{U}\cup\mathbb{S}} w_{ij}^{l+1} x_j^l + b_i^{l+1} = \sum_{j\in\mathbb{U}\cup\mathbb{T}} w_{ij}^{l+1} x_j^l + \sum_{j\in\mathbb{S}\setminus\mathbb{T}} w_{ij}^{l+1}\left(\sum_{k\in\mathbb{T}} \alpha_k^j x_k^l + \left(b_j^l - \sum_{k\in\mathbb{T}} \alpha_k^j b_k^l\right)\right) + b_i^{l+1} = \sum_{j\in\mathbb{U}} w_{ij}^{l+1} x_j^l + \sum_{j\in\mathbb{T}} \left(w_{ij}^{l+1} + \sum_{k\in\mathbb{S}\setminus\mathbb{T}} \alpha_j^k w_i^{l+1} k\right) x_j^l + \left(b_i^{l+1} + \sum_{j\in\mathbb{S}\setminus\mathbb{T}} w_{ij}^{l+1}\left(b_j^l - \sum_{k\in\mathbb{T}} \alpha_k^j b_k^l\right)\right)$. □

In Algorithm 1, we use relationships implied by the proof of Corollary 4 with $\mathbb{S} = \mathbb{Q}^l$ and $\mathbb{T} = \overline{\mathbb{Q}}$ to merge stably active neurons. By adjusting the biases of the neurons in the next layer as well as the weights connecting every neuron in $\overline{\mathbb{Q}}$ with the neurons in the next layer, we could set a weight of 0 to the connections between every neuron in $\mathbb{Q}^l \setminus \overline{\mathbb{Q}}$ and the neurons in the next layer. Hence, we simply remove all neurons in $\mathbb{Q}^l \setminus \overline{\mathbb{Q}}$ after adjusting those network parameters.

The case in which an entire layer is stably active—either before any compression is applied or once stably inactive neurons are removed—is also considered separately.

Folding of stable layers This operation is performed between lines 26 and 29. We use the following results to show that stable layers can be folded in a single step. We refer to Appendix C for the proofs of these results.

Proposition 5. *If all the neurons of layer $l \in \mathbb{L} \setminus \{L\}$ are stably active, then the pre-activation output of layer $l+1$ is an affine function on the inputs of layer l.*

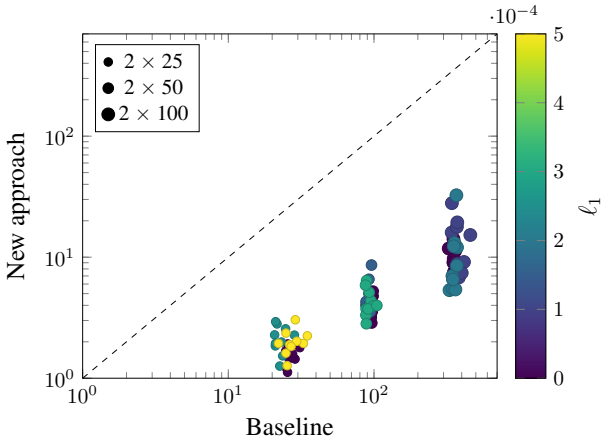

Figure 1. Runtime (in seconds) to identify stable neurons using the new approach vs. the baseline from Serra et al. (2020).

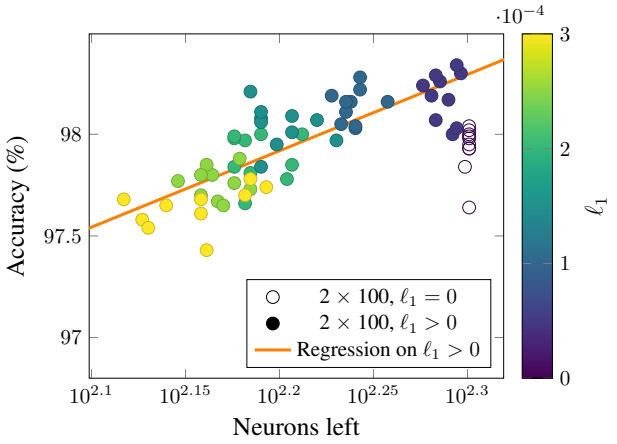

Figure 2. Relationship between the number of neurons in hidden layers after exact compression and the accuracy of neural networks trained with same number of nodes and different ℓ_1 regularization.

Corollary 6. *If all neurons of layer $l \in \mathbb{L} \setminus \{L\}$ are stable, then the pre-activation output of layer $l+1$ is an affine function on the inputs of layer l.*

In Algorithm 1, we use relationships implied by the proof of Corollary 6 with $\mathbb{S} = \mathbb{Q}^l$ to fold stable layers. By adjusting the biases as well as the incoming weights of layer $l+1$, we directly use the inputs of layer $l-1$ instead.

Although the steps above would apply if a layer is stably inactive, that case deserves separate consideration.

Collapse of a network with stably inactive layers This operation is performed between lines 20 and 25. If layer $l \in \mathbb{L}$ are stably inactive, then $\boldsymbol{x}^l = 0$ for any input $\boldsymbol{x}^0 \in \mathbb{X}$ and thus the value of $\boldsymbol{x}^L$ is constant. Hence, we collapse layers 1 to $L-1$ by making the output of the remaining layer constant and equal to such value of $\boldsymbol{x}^L$.

5.4. On the Complexity of the New Algorithm

While our algorithm requires solving fewer optimization problems than in Serra et al. (2020), the dependence on solving a single NP-hard problem—such as MILP formulations in general—implies an exponential worst-case complexity. Nevertheless, the progress of MILP in the past decades makes it possible to solve considerably large problems with state-of-art MILP solvers. In that context, we show the performance gains of our algorithm empirically.

6. Experimental Results

We trained and evaluated the compressibility of classifiers and autoencoders for the MNIST dataset (LeCun et al., 1998) with and without ℓ_1 weight regularization, which is known to induce stability (Tjeng et al., 2019). We refer to Appendix D for details on implementation and environment.

For the classifiers, we use the notation $L \times n$ for the architecture of L hidden layers with n neurons each. We started at $L = 2$ and $n = 100$, and then doubled the width n or incremented the depth L until the majority of the runs for any configuration timed out after 7,200 seconds. For each architecture, we identified an amount $\ell_1 = \bar{\ell}$ of regularization during training that produces the same accuracy as training with $\ell_1 = 0$. We trained and evaluated neural networks with 10 different random initialization seeds using $\ell_1 = 0$, $\ell_1 = 0.5\bar{\ell}$, and $\ell_1 = \bar{\ell}$. The amount of regularization used did not stabilize the entire layer. Table 1 reports the runtime to identify stable neurons and the proportion of neurons—as well as the corresponding connections—that can be removed due to stability in each case. Additional experiments with other architectures, configurations, and formulations are discussed along the analysis.

Runtime improvement Figure 1 compares the baseline (Serra et al., 2020) with our approach on smaller architectures—2×25, 2×50, and 2×100—using ℓ_1 as described above. The median ratio between runtimes is 21.

When the baseline is compared with only using the new formulation, the median runtime ratio is 12. When using the new formulation is compared with also using the induced solutions, the median runtime ratio is 1.6. We refer to Appendix E for figures involving the latter two comparisons.

The overall speedup is greater in larger networks: the median runtime ratio is 15 for 2×25 and 39 for 2×100.

Effect of regularization on compressibility We observe more compression with more ℓ_1 regularization. For sufficiently large networks having the same accuracy as those trained with $\ell_1 = 0$, we can remove around 20% of the neurons and 40% of the connections. In line with Serra

Table 1. Compression results for classifiers trained on MNIST with varying architectures and levels of regularization.

Architecture	ℓ_1	Accuracy (%)	Compression Runtime (s)	% Removed Neurons	% Removed Connections	Timed Out (Out of 10)
2×100	0	97.93±0.04	10.1±0.7	0.05±0.05	0.1±0.1	0
2×100	0.0001	98.14±0.03	14±2	13.2±0.5	23.4±0.9	0
2×100	0.0002	97.89±0.04	11±3	22.7±0.8	38±1	0
2×200	0	98.17±0.02	45±7	0.02±0.02	0.05±0.05	0
2×200	0.00006	98.33±0.01	120±20	12.8±0.4	23.3±0.8	0
2×200	0.00012	98.17±0.02	26±2	25.8±0.4	44.2±0.6	0
2×400	0	98.25±0.03	400±30	0.02±0.02	0.05±0.05	0
2×400	0.00004	98.35±0.02	1800±100	8.6±0.2	16.3±0.4	0
2×400	0.00008	98.24±0.02	130±10	24.5±0.3	42.7±0.5	0
2×800	0	98.28±0.02	3500±600	0.02±0.01	0.05±0.02	0
2×800	0.000027	—	—	—	—	10
2×800	0.000054	98.29±0.02	1000±100	21.6±0.3	38.5±0.5	0
3×100	0	98.05±0.02	54±3	0±0	0±0	0
3×100	0.0001	98.23±0.02	80±10	14.2±0.5	25.1±0.9	0
3×100	0.0002	98.05±0.04	50±20	25.6±0.8	42±1	0
4×100	0	98.12±0.01	220±20	0.1±0.07	0.2±0.1	0
4×100	0.0001	98.18±0.07	2000±1000	15.6±0.9	28±2	5
4×100	0.0002	98.12±0.03	180±50	25.3±0.6	42.5±1	0
5×100	0	98.13±0.03	841.1±0.1	0.2±0.1	0.3±0.2	1
5×100	0.0001	98.42	3369.5	16.8	30.8	9
5×100	0.0002	98.12±0.03	511.3±0.5	27.3±0.3	46.2±0.6	0

et al. (2020), we observe that the exact compressibility of neural networks trained with $\ell_1 = 0$ is negligible, but also that you can have the cake and eat it too: training with $\ell_1 = 0.5\overline{\ell}$ leads to better accuracy and a smaller network.

Runtime variability Based on how the runtimes vary depending on regularization, we note that the the configuration with $\ell_1 = 0.5\overline{\ell}$ takes more time to solve, especially in larger networks. Curiously, however, the compression runtimes for networks trained with $\ell_1 = \overline{\ell}$ are smaller than for those with $\ell_1 = 0$, which suggests that compression runtimes first increase and then decrease with regularization.

Relationship between compressibility and accuracy Figure 2 analyzes the relationship between classifier accuracy and the number of neurons left after compression. We fix the architecture to 2×100 while also training and evaluating neural networks with 10 different seeds using from $\ell_1 = 1.5\overline{\ell}$ to $\ell_1 = 3\overline{\ell}$ in steps of $0.5\overline{\ell}$, hence moving beyond the point at which regularization helps. When excluding $\ell_1 = 0$, we obtain a linear regression on the semilog plot with coefficient of determination $R^2 = 0.64$. That suggests that the accuracy is a good proxy for how much a neural network trained with ℓ_1 regularization can be compressed.

An application to autoencoders We explored more extreme cases of compression with autoencoders, in which the variation in layer width may intuitively favor merging stably active neurons and folding stable layers. These experiments involved architectures with $L = 3$ and total number of neurons between 100 and 400, for which the average loss without regularization is 0.04. We refer to Appendix F for more details. With $\ell_1 = 0.00002$, the average loss is 0.047, the compression takes 460 seconds, and it is possible to remove 15% of the neurons and 10% of the connections. With $\ell_1 = 0.0002$, the average loss is 0.078, the compression takes 4 seconds, and it is possible to remove nearly all neurons in the hidden layers. Hence, we can quickly identify special cases in which large autoencoders trained with a sufficient amount of ℓ_1 regularization can be exactly replaced by a linear function, which happens even if the extra regularization entails only marginal reconstruction loss.

7. Conclusion

This paper outlined the potential for exact compression of neural networks and presented an approach that makes it practical for sizes that are large enough for many applications. To the best of our knowledge, our approach is the state-of-the-art for optimization-based exact compression.

Our performance improvements come from insights about the MILP formulations associated with optimization problems over neural networks, which have many other applications besides exact compression. Most notably, such formulations are also used for network verification (Bunel et al., 2017; Liu et al., 2019a; Rössig & Petkovic, 2020).

References

Aghasi, A., Abdi, A., Nguyen, N., and Romberg, J. NetTrim: Convex pruning of deep neural networks with performance guarantee. In *NeurIPS*, 2017.

Al-Falou, A. and Trummer, D. Identifiability of recurrent neural networks. *Econometric Theory*, 19(5):812–828, 2003.

Albertini, F. and Sontag, E. For neural networks, function determines form. *Neural Networks*, 6(7):975–990, 1993a.

Albertini, F. and Sontag, E. Identifiability of discrete-time neural networks. In *ECC*, 1993b.

Amodei, D., Hernandez, D., Sastry, G., Clark, J., Brockman, G., and Sutskever, I. AI and compute. https://openai.com/blog/ai-and-compute/, 2018. Accessed: 2020-12-23.

Anderson, R., Huchette, J., Tjandraatmadja, C., and Vielma, J. Strong mixed-integer programming formulations for trained neural networks. In *IPCO*, 2019.

Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., and Vielma, J. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming*, 183:3–39, 2020.

Arora, S., Ge, R., Neyshabur, B., and Zhang, Y. Stronger generalization bounds for deep nets via a compression approach. In *ICML*, 2018.

Bergman, D., Huang, T., Brooks, P., Lodi, A., and Raghunathan, A. JANOS: An integrated predictive and prescriptive modeling framework. *arXiv*, 1911.09461, 2019.

Berner, J., Elbrächter, D., and Grohs, P. How degenerate is the parametrization of neural networks with the relu activation function? In *NeurIPS*, 2019.

Blalock, D., Ortiz, J., Frankle, J., and Guttag, J. What is the state of neural network pruning? In *MLSys*, 2020.

Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., and Misener, R. Efficient verification of relu-based neural networks via dependency analysis. In *AAAI*, 2020.

Bridle, J. S. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pp. 227–236. 1990.

Bunel, R., Turkaslan, I., Torr, P. H. S., Kohli, P., and Kumar, M. P. Piecewise linear neural network verification: A comparative study. *CoRR*, abs/1711.00455, 2017.

Büning, M. K., Kern, P., and Sinz, C. Verifying equivalence properties of neural networks with relu activation functions. In *CP*, 2020.

Chen, A. M., Lu, H., and Hecht-Nielsen, R. On the geometry of feedforward neural network error surfaces. *Neural Computation*, 5(6):910–927, 1993.

Cheng, C., Nührenberg, G., and Ruess, H. Maximum resilience of artificial neural networks. In *ATVA*, 2017.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1):126–136, 2018.

Cook, S. The complexity of theorem-proving procedures. In *STOC*, 1971.

Dantzig, G., Fulkerson, D., and Johnson, S. Solution of a large scale traveling salesman problem. Technical Report P-510, RAND Corporation, Santa Monica, California, USA, 1954.

Delarue, A., Anderson, R., and Tjandraatmadja, C. Reinforcement learning with combinatorial actions: An application to vehicle routing. In *NeurIPS*, 2020.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. Predicting parameters in deep learning. In *NeurIPS*, 2013.

Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. In *NeurIPS*, 2014.

Dong, X., Chen, S., and Pan, S. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *NeurIPS*, 2017.

ElAraby, M., Wolf, G., and Carvalho, M. Identifying efficient sub-networks using mixed integer programming. In *OPT Workshop*, 2020.

Fefferman, C. and Markel, S. Recovering a feed-forward net from its output. In *NeurIPS*, 1994.

Fischetti, M. and Jo, J. Deep neural networks and mixed integer linear optimization. *Constraints*, 23:296–309, 2018.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2019.

Frankle, J., Dziugaite, G., Roy, D., and Carbin, M. Pruning neural networks at initialization: Why are we missing the mark? In *ICLR*, 2021.

Gordon, M., Duh, K., and Andrews, N. Compressing BERT: Studying the effects of weight pruning on transfer learning. In *Rep4NLP Workshop*, 2020.

Gurobi Optimization, L. Gurobi optimizer reference manual, 2020. URL https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.1/refman.pdf. Version 9.1.

Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., and Seung, S. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947–951, 2000.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

Han, S., Mao, H., and Dally, W. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*, 2016.

Hanin, B. and Rolnick, D. Complexity of linear regions in deep networks. In *ICML*, 2019a.

Hanin, B. and Rolnick, D. Deep ReLU networks have surprisingly few activation patterns. In *NeurIPS*, 2019b.

Hanson, S. and Pratt, L. Comparing biases for minimal network construction with back-propagation. In *NeurIPS*, 1988.

Hassibi, B., Stork, D., and Wolff, G. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, 1993.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017.

Hecht-Nielsen, R. On the algebraic structure of feedforward network weight spaces. In Eckmiller, R. (ed.), *Advanced Neural Computers*, pp. 129–135. North-Holland, 1990.

Hooker, S., Courville, A., Clark, G., Dauphin, Y., and Frome, A. What do compressed deep neural networks forget? *arXiv*, 1911.05248, 2019.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014.

Janowsky, S. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600–6603, 1989.

Kůrková, V. and Kainen, P. Functionally equivalent feedforward neural networks. *Neural Computation*, 6(3):543–558, 1994.

Kumar, A., Serra, T., and Ramalingam, S. Equivalent and approximate transformations of deep neural networks. *arXiv*, 1905.11428, 2019.

Lebedev, V. and Lempitsky, V. Fast ConvNets using group-wise brain damage. In *CVPR*, 2016.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *ICLR*, 2015.

LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. In *NeurIPS*, 1989.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Lee, N., Ajanthan, T., and Torr, P. SNIP: Single-shot network pruning based on connection sensitivity. In *ICLR*, 2019.

Lee, N., Ajanthan, T., Gould, S., and Torr, P. A signal propagation perspective for pruning neural networks at initialization. In *ICLR*, 2020.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. Pruning filters for efficient convnets. In *ICLR*, 2017.

Liu, C., Arnon, T., Lazarus, C., Barrett, C. W., and Kochenderfer, M. J. Algorithms for verifying deep neural networks. *CoRR*, abs/1903.06758, 2019a.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. In *ICLR*, 2019b.

Luo, J.-H., Wu, J., and Lin, W. ThiNet: A filter level pruning method for deep neural network compression. In *ICCV*, 2017.

Mariet, Z. and Sra, S. Diversity networks: Neural network compression using determinantal point processes. In *ICLR*, 2016.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. In *ICLR*, 2017.

Montúfar, G., Pascanu, R., Cho, K., and Bengio, Y. On the number of linear regions of deep neural networks. In *NeurIPS*, 2014.

Mozer, M. and Smolensky, P. Using relevance to reduce network size automatically. *Connection Science*, 1(1): 3–16, 1989.

Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., and Walsh, T. Verifying properties of binarized deep neural networks. In *AAAI*, 2018.

Pascanu, R., Montúfar, G., and Bengio, Y. On the number of response regions of deep feedforward networks with piecewise linear activations. In *ICLR*, 2014.

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

Petersen, P., Raslan, M., and Voigtlaender, F. Topological properties of the set of functions generated by neural networks of fixed size. *Foundations of Computational Mathematics*, 2020.

Phuong, M. and Lampert, C. Functional vs. parametric equivalence of ReLU networks. In *ICLR*, 2020.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Dickstein, J. On the expressive power of deep neural networks. In *ICML*, 2017.

Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. In *ICLR*, 2020.

Rolnick, D. and Kording, K. Reverse-engineering deep ReLU networks. In *ICML*, 2020.

Rosenfeld, J., Frankle, J., Carbin, M., and Shavit, N. On the predictability of pruning across scales. *arXiv*, 2006.10621, 2020.

Rössig, A. and Petkovic, M. Advances in verification of ReLU neural networks. *Journal of Global Optimization*, 2020.

Say, B., Wu, G., Zhou, Y. Q., and Sanner, S. Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programming. In *IJCAI*, 2017.

Serra, T. and Ramalingam, S. Empirical bounds on linear regions of deep rectifier networks. In *AAAI*, 2020.

Serra, T., Tjandraatmadja, C., and Ramalingam, S. Bounding and counting linear regions of deep neural networks. In *ICML*, 2018.

Serra, T., Kumar, A., and Ramalingam, S. Lossless compression of deep neural networks. In *CPAIOR*, 2020.

Sourek, G. and Zelezny, F. Lossless compression of structured convolutional models via lifting. In *ICLR*, 2021.

Srinivas, S. and Babu, R. V. Data-free parameter pruning for deep neural networks. In *BMVC*, 2015.

Suau, X., Zappella, L., and Apostoloff, N. Filter distillation for network compression. In *WACV*, 2020.

Suzuki, T., Abe, H., Murata, T., Horiuchi, S., Ito, K., Wachi, T., Hirai, S., Yukishima, M., and Nishimura, T. Spectral-pruning: Compressing deep neural network via spectral analysis. In *IJCAI*, 2020a.

Suzuki, T., Abe, H., and Nishimura, T. Compression based bound for non-compressed network: Unified generalization error analysis of large compressible deep neural network. In *ICLR*, 2020b.

Tanaka, H., Kunin, D., Yamins, D., and Ganguli, S. Pruning neural networks without any data by iteratively conserving synaptic flow. In *NeurIPS*, 2020.

Tjeng, V., Xiao, K., and Tedrake, R. Evaluating robustness of neural networks with mixed integer programming. In *ICLR*, 2019.

Wang, C., Grosse, R., Fidler, S., and Zhang, G. EigenDamage: Structured pruning in the Kronecker-factored eigenbasis. In *ICML*, 2019.

Wang, C., Zhang, G., and Grosse, R. Picking winning tickets before training by preserving gradient flow. In *ICLR*, 2020.

Xiao, K., Tjeng, V., Shafiullah, N., and Madry, A. Training for faster adversarial robustness verification via inducing ReLU stability. *ICLR*, 2019.

Xing, X., Sha, L., Hong, P., Shang, Z., and Liu, J. Probabilistic connection importance inference and lossless compression of deep neural networks. In *ICLR*, 2020.

Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V., Han, X., Gao, M., Lin, C.-Y., and Davis, L. NISP: Pruning networks using neuron importance score propagation. In *CVPR*, 2018.

Zeng, W. and Urtasun, R. MLPrune: Multi-layer pruning for automated neural network compression. 2018.

Zhou, W., Veitch, V., Austern, M., Adams, R., and Orbanz, P. Non-vacuous generalization bounds at the ImageNet scale: A PAC-Bayesian compression approach. In *ICLR*, 2019.

Supplementary Material

A. On Dropping Constraint (19)

We avoid explicitly enforcing that variables p_i^l and q_i^l are binary by leveraging that z_i^l is binary. Constraint (17) implies that $p_i^l \in [0,1]$ and $p_i^l \neq 0$ only if $z_i^l = 1$. In turn, if $z_i^l = 1$, then we can assume $p_i^l = 1$ by optimality since the objective function (14) maximizes the sum of those variables and no other constraint limits its value. Likewise, constraint (18) implies that $q_i^l \in [0,1]$ and $q_i^l \neq 0$ only if $z_i^l = 0$. In turn, if $z_i^l = 0$, then likewise we can assume $q_i^l = 1$ by optimality since the objective function (14) maximizes the sum of those variables and no other constraint limits its value. Reducing the number of binary variables is widely regarded as a good practice to make MILP formulations easier to solve.

B. On Lazy Constraint Callbacks

Lazy constraint callbacks are generally used when the total number of constraints of an MILP formulation is prohibitively large. One such example is the most commonly used formulation for the traveling salesperson problem due to the subtour elimination constraints (Dantzig et al., 1954). The callback allows us to handle such cases more efficiently by formulating the problem with fewer constraints and then adding the remaining ones only if they are necessary to rule out infeasible solutions. Every time that a supposedly feasible solution is found, the MILP solver invokes the callback implemented by the user for an opportunity to make such a solution infeasible by adding one of the missing constraints that the supposedly feasible solution does not satisfy. If none is provided by the callback, the MILP solver accepts the solution as feasible.

In our case, we use a lazy constraint callback for a slightly different purpose. Namely, we implement the callback to (i) inspect every feasible solution that is obtained; and (ii) mimic the updates that would have been made to $\boldsymbol{P}$ and $\boldsymbol{Q}$ between consecutive calls to the solver by adding constraints that set the value of either p_i^l or q_i^l to zero once a solution is found in which such variable has a positive value. In other words, the callback adds constraints to ignore the effect of p_i^l or q_i^l on the objective function if we know that the i-th neuron of layer l is active or inactive for some input, respectively. Therefore, the MILP solver will eventually produce an optimal solution of value zero once the set of solutions inspected by the callback covers all the possible states for the neurons and the remaining states are deemed unattainable after an exhaustive search.

C. Proofs of Proposition 5 and Corollary 6

Proposition 5. *If all the neurons of layer $l \in \mathbb{L} \setminus \{L\}$ are stably active, then the pre-activation output of layer $l+1$ is an affine function on the inputs of layer l.*

Proof. Since $\boldsymbol{x}^l = \boldsymbol{W}^l \boldsymbol{x}^{l-1} + \boldsymbol{b}^l$, then $\boldsymbol{y}^{l+1} = \boldsymbol{W}^{l+1}\boldsymbol{x}^l + \boldsymbol{b}^{l+1} = \boldsymbol{W}^{l+1}\boldsymbol{W}^l\boldsymbol{x}^{l-1} + \left(\boldsymbol{W}^{l+1}\boldsymbol{b}^l + \boldsymbol{b}^{l+1}\right)$. □

Corollary 6. *If all neurons of layer $l \in \mathbb{L} \setminus \{L\}$ are stable, then the pre-activation output of layer $l+1$ is an affine function on the inputs of layer l.*

Proof. Let $\mathbb{S}$ be the set of stably active neurons in layer l. If $|\mathbb{S}| < n_l$, the identity $\boldsymbol{x}^l = \boldsymbol{W}^l\boldsymbol{x}^{l-1} + \boldsymbol{b}^l$ still holds if the bias and the weights of all the connections of the neurons not in $\mathbb{S}$ with the neurons in the next layer are 0. More generally, we can thus obtain an equivalent neural network if $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$ are both premultiplied by $\boldsymbol{I}_{n_l}(\mathbb{S})$ since that only would change the weights and biases associated with the neurons not in $\mathbb{S}$ to 0. Hence, the identity $\boldsymbol{x}^l = \boldsymbol{I}_{n_l}(\mathbb{S})\left(\boldsymbol{W}^l\boldsymbol{x}^{l-1} + \boldsymbol{b}^l\right)$ always holds if all neurons in layer l are stable, which implies that $\boldsymbol{y}^{l+1} = \boldsymbol{W}^{l+1}\boldsymbol{I}_{n_l}(\mathbb{S})\boldsymbol{W}^l\boldsymbol{x}^{l-1} + \left(\boldsymbol{W}^{l+1}\boldsymbol{I}_{n_l}(\mathbb{S})\boldsymbol{b}^l + \boldsymbol{b}^{l+1}\right)$. □

D. Details on the Computational Experiments

We implemented the fully connected architectures in Pytorch (Paszke et al., 2019). All these networks have input size 784 and ReLU activations but have varying hidden layers and hidden units per layer depending on the experiments. The output units are kept at 10 and 784 units for classification and autoencoders respectively. For the classification, the output is passed through a softmax layer and binary cross-entropy as the loss function. For the autoencoders, MSE loss is used as the loss function. No augmentation of training images is done.

The weights of the network are initialized with the Kaiming initialization (He et al., 2015), and the biases are initialized to zero. Training proceeds from scratch takes 120 epochs and starts with learning rate of 0.01, which is decayed by a factor of 0.1 after every 50 epochs. SGD with momentum optimizer is used, with a momentum of 0.9 and batch size 64. Unless stated otherwise, we use ℓ_1 regularization whose actual value depends on the experiments. The weight decay is kept at 0 unless otherwise stated. We consider the model saved in the last epoch as our final model.

We solve the MILP formulations using Gurobi 9.1.0 through gurobipy (Gurobi Optimization, 2020). The value of the positive constants M_i^l and μ_i^l for each neuron are calculated

with an upper bound of on the values of x_i^l and χ_i^l through interval arithmetic by taking element-wise maxima (Cheng et al., 2017).

The classifier experiments were run on a machine with Intel(R) Core(TM) i5-8365U CPU @ 1.60 GHz and 16 GB of RAM. The autoencoder experiments were run on a machine with 40 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz processors, 126 GB of RAM, and one 12GB Nvidia Titan Xp GPU.

The results reported in Tables 1 and 2 contain the mean and the standard error with respect to the runs that did not time out for each configuration tested.

E. Additional Figures

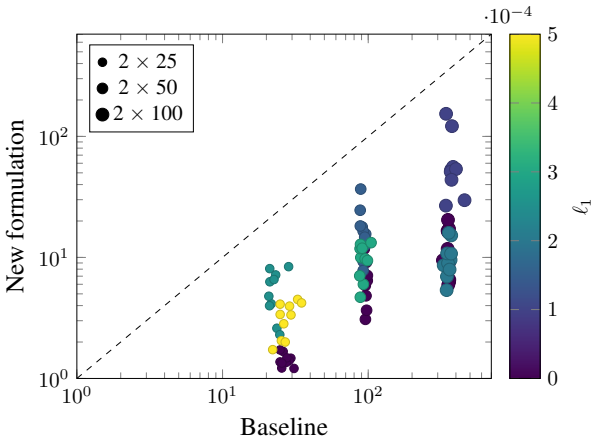

Figure 3. Runtime (in seconds) to identify stable neurons using the new formulation vs. multiple formulations as in prior work.

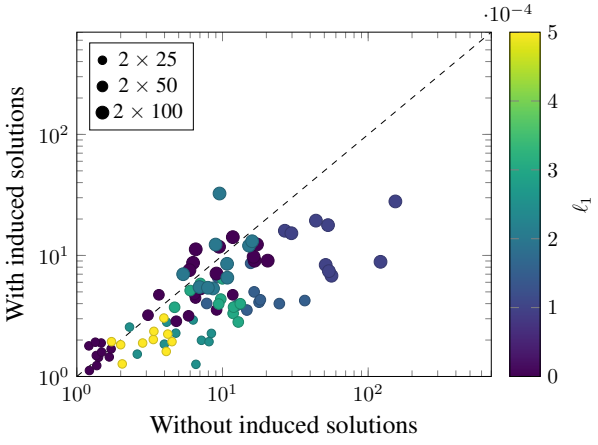

Figure 4. Runtime (in seconds) to identify stable neurons using the new formulation with and without solutions induced by linear relaxation.

F. Autoencoder

For the autoencoders, We use the notation $n_1 \mid n_2 \mid n_3$ for the architecture of 3 hidden layers with n_1, n_2, and n_3 neurons. The output layer has the same size as the input, 784, and uses ReLU activation. Starting with the architecture $100 \mid 10 \mid 100$, we evaluated changes to the bottleneck width n_2 as well as to the width of the other two layers. First, we changed the bottleneck width to $n_2 = 25$ and $n_2 = 50$. Second, we changed the width of the other layers to $n_1, n_3 = 50$, $n_1, n_3 = 200$, and $n_1, n_3 = 400$ while keeping $n_2 = 10$. For each architecture, we trained and evaluated neural networks with 5 different random initialization seeds using $\ell_1 = 0$, $\ell_1 = 0.00002$, and $\ell_1 = 0.0002$. Table 2 reports the runtime to identify stable neurons and the proportion of neurons—as well as the corresponding connections—that can be removed due to stability in each case.

With the largest amount of regularization, we notice that the runtimes are considerably smaller and most of the network can be removed while the loss during training only doubles in comparison to using zero or a moderate amount of regularization. In fact, the only neurons that are not stable in such case are in the first layer, whereas between 3 and 4 out of the 5 neural networks trained for each architecture have all hidden layers completely stable. By also evaluating the stability of the output layer, we identified a few cases in which the output layer is entirely stable. While we have not explicitly explored that possibility in the proposed algorithm, the implication for such case is that the neural network can be reduced to a linear function on the domain of interest. With autoencoders, we observed that this can happen when the regularization during training no more than doubles the loss, and that we can evaluate if that happens within seconds: the runtime when the stability of the output layer is tested is 12 seconds on average and never more than 25 seconds.

Figure 5 shows the difference in runtimes between our approach and the baseline (Serra et al., 2020) for higher regularization, fixed $n_2 = 10$, and varying but equal values for n_1 and n_3. In this particular case, we observe that the new method presents a considerable gain in performance, which increases with the width of the non-bottleneck layers.

Table 2. Compression results for autoencoders trained on MNIST with varying architectures and levels of regularization.

Architecture	ℓ_1	Loss	Compression Runtime (s)	% Removed Neurons	% Removed Connections	Timed Out (Out of 5)
100 | 10 | 100	0	0.045 ± 0.001	130 ± 30	0.1 ± 0.1	0.05 ± 0.06	0
100 | 10 | 100	0.00002	0.047 ± 0.0009	120 ± 30	12.7 ± 0.6	7.2 ± 0.9	0
100 | 10 | 100	0.0002	0.077 ± 0.002	2.73 ± 0.05	95 ± 6	90 ± 10	0
100 | 25 | 100	0	0.035 ± 0.001	500 ± 300	0 ± 0	0 ± 0	0
100 | 25 | 100	0.00002	0.047 ± 0.001	800 ± 200	14 ± 1	10 ± 2	0
100 | 25 | 100	0.0002	0.076 ± 0.001	2.88 ± 0.08	90 ± 7	80 ± 20	0
100 | 50 | 100	0	0.0311 ± 0.0009	230 ± 20	0 ± 0	0 ± 0	0
100 | 50 | 100	0.00002	0.0478 ± 0.0009	600 ± 200	17.4 ± 0.9	13 ± 1	0
100 | 50 | 100	0.0002	0.081 ± 0.003	2.96 ± 0.04	90 ± 7	80 ± 20	0
50 | 10 | 50	0	0.047 ± 0.002	33 ± 4	0 ± 0	0 ± 0	0
50 | 10 | 50	0.00002	0.051 ± 0.002	50 ± 20	14 ± 3	13 ± 2	0
50 | 10 | 50	0.0002	0.081 ± 0.002	1.42 ± 0.02	89 ± 8	88 ± 8	0
100 | 10 | 100	0	0.045 ± 0.001	130 ± 30	0.1 ± 0.1	0.05 ± 0.06	0
100 | 10 | 100	0.00002	0.047 ± 0.0009	120 ± 30	12.7 ± 0.6	7.2 ± 0.9	0
100 | 10 | 100	0.0002	0.077 ± 0.002	2.73 ± 0.05	95 ± 6	90 ± 10	0
200 | 10 | 200	0	0.041 ± 0.002	1000 ± 1000	0.4 ± 0.4	0.4 ± 0.4	1
200 | 10 | 200	0.00002	0.043 ± 0.002	700 ± 400	14 ± 0.7	7 ± 1	0
200 | 10 | 200	0.0002	0.076 ± 0.002	5.41 ± 0.03	95 ± 6	80 ± 20	0
400 | 10 | 400	0	0.04	2704	0	0	4
400 | 10 | 400	0.00002	0.0395 ± 0.001	1300 ± 100	15 ± 1	6 ± 0.7	0
400 | 10 | 400	0.0002	0.073 ± 0.001	10.5 ± 0.2	89.1 ± 7.5	13.6 ± 59.3	0

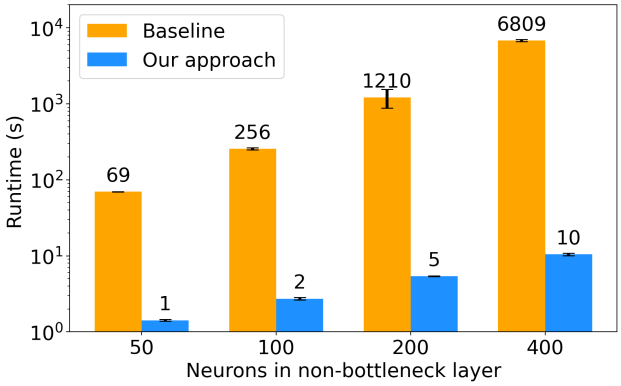

Figure 5. Runtime (in seconds) to identify stable neurons in autoencoders with high regularization ($\ell_1 = 0.0002$) using the baseline approach with multiple formulations vs. our approach.