Constraints (2018) 23:296–309
https://doi.org/10.1007/s10601-018-9285-6

Deep neural networks and mixed integer linear optimization

Matteo Fischetti[1] · **Jason Jo**[2,3]

Published online: 26 April 2018

Abstract Deep Neural Networks (DNNs) are very popular these days, and are the subject of a very intense investigation. A DNN is made up of layers of internal units (or neurons), each of which computes an affine combination of the output of the units in the previous layer, applies a nonlinear operator, and outputs the corresponding value (also known as activation). A commonly-used nonlinear operator is the so-called rectified linear unit (ReLU), whose output is just the maximum between its input value and zero. In this (and other similar cases like max pooling, where the max operation involves more than one input value), for fixed parameters one can model the DNN as a 0-1 Mixed Integer Linear Program (0-1 MILP) where the continuous variables correspond to the output values of each unit, and a binary variable is associated with each ReLU to model its yes/no nature. In this paper we discuss the peculiarity of this kind of 0-1 MILP models, and describe an effective bound-tightening technique intended to ease its solution. We also present possible applications of the 0-1 MILP model arising in feature visualization and in the construction of adversarial examples. Computational results are reported, aimed at investigating (on small DNNs) the computational performance of a state-of-the-art MILP solver when applied to a known test case, namely, hand-written digit recognition.

This article belongs to the Topical Collection: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*
Guest Editor: Willem-Jan van Hoeve

✉ Matteo Fischetti
matteo.fischetti@unipd.it

Jason Jo
jason.jo.research@gmail.com

[1] Department of Information Engineering (DEI), University of Padova, Padua, Italy

[2] Montreal Institute for Learning Algorithms (MILA), Montreal, Québec, Canada

[3] Institute for Data Valorization (IVADO), Montreal, Québec, Canada

Keywords Deep neural networks · Mixed-integer programming · Deep learning · Mathematical optimization · Computational experiments

1 Introduction

Deep Neural Networks (DNNs) are among the most popular and effective Machine Learning (ML) architectures and are the subject of a very intense investigation; see e.g. [8]. A DNN is made up of layers of internal units (also known as *neurons*), each of which computes an affine combination of the output of the units in the previous layer, applies a nonlinear operator, and outputs the corresponding value (also known as *activation*). A commonly-used nonlinear operator is the so-called *rectified linear unit* (ReLU) [11], whose output is just the maximum between its input value and zero.

In this work we address a DNN with ReLU and/or max (or average) pooling activations. We investigate a 0-1 Mixed-Integer Linear Programming (0-1 MILP) model of the DNN when its parameters are fixed, and highlight some of its peculiarities. We also computationally analyze a bound-tightening mechanism that has a relevant impact in reducing solution times. Two applications of the 0-1 MILP model in the context of *feature visualization* [4] and *adversarial machine learning* [14] are outlined, the latter being very well suited for our approach as finding (almost) optimal solutions is an important research topic.

The present paper is organized as follows. In Section 2 we give a step-by-step derivation of a 0-1 MILP model that describes the computation performed by a given DNN with ReLU activations (and fixed parameters) to compute the DNN output as a function of its input. We also briefly outline some of the peculiarities of this model. As all DNN parameters are assumed to be fixed, the model (as stated) cannot be used for training. Alternative applications of the model (namely, feature visualization and construction of adversarial examples) are discussed in Section 3. Section 4 addresses the computational performance of a state-of-the art commercial MILP solver (`IBM ILOG CPLEX 12.7`) in solving the 0-1 MILP instances arising when constructing optimal adversarial examples for a known test case, namely, hand-written digit recognition. The results show that, for small DNNs, these instances can typically be solved to proven optimality in a matter of seconds/minutes on a standard notebook. Some conclusions and directions of future work are finally addressed in Section 5.

An early version of the present paper was submitted to the CPAIOR 2018 conference in November 2017. We recently became aware that a 0-1 MILP model similar to the one studied in the present paper, has been independently proposed (almost at the same time) in [2, 13, 15]. Therefore we cannot claim the model is new. However, to the best of our knowledge the applications and discussions we report in the present paper are original and hopefully of interest for both the Mathematical Optimization and the Machine Learning communities.

2 A 0-1 MILP model

Let the DNN be made up of $K+1$ (say) layers, numbered from 0 to K. Layer 0 is fictitious and corresponds to the input of the DNN, while the last layer, K corresponds to its outputs. Each layer $k \in \{0, 1, \ldots, K\}$ is made up of n_k units (i.e., nodes in networks, or neurons), numbered from 1 to n_k. We denote by $\texttt{UNIT}(j, k)$ the j-th unit of layer k.

Let $x^k \in \Re^{n_k}$ be the output vector of layer k, i.e., x^k_j is the output of $\texttt{UNIT}(j,k)$ ($j = 1,\ldots,n_k$). As already mentioned, layer 0 corresponds to the DNN input, hence x^0_j is the j-th input value (out of n_0) for the DNN. Analogously, x^K_j is the j-th output value (out of n_K) of the DNN viewed as a whole. For each layer $k \geq 1$, $\texttt{UNIT}(j,k)$ computes its output vector x^k through the formula

$$x^k = \sigma(W^{k-1}x^{k-1} + b^{k-1}),$$

where $\sigma(\cdot)$ is a nonlinear function (possibly depending on j and k), and W^{k-1} (resp. b^{k-1}) is a *given* matrix of weights (resp., vector of biases).

As in many applications, we will assume that σ is a *rectified linear unit*, i.e., the equations governing the DNN are

$$x^k = ReLU(W^{k-1}x^{k-1} + b^{k-1}), \quad k = 1,\ldots,K \tag{1}$$

where, for a real vector y, $ReLU(y) := \max\{0, y\}$ (componentwise).

Note that the weight/bias matrices (W, b) can contain negative entries, while all the output vectors x^k are nonnegative, with the possible exception of the vector x^0 that represents the input of the DNN as a whole.

To get a valid 0-1 MILP model for a given DNN, one needs to study the basic scalar equation

$$x = ReLU(w^T y + b).$$

To this end, one can write the linear conditions

$$w^T y + b = x - s, \quad x \geq 0, \quad s \geq 0 \tag{2}$$

to decouple the positive and negative part of the ReLU input. Unfortunately, the solution (x, s) of constraints (2) is not unique (as it should be because ReLU() is in fact a function), because one can always take any scalar $\delta \geq 0$ and obtain a still-feasible solution $(x+\delta, s+\delta)$. Imposing $\delta = 0$ is therefore the only critical issue to be addressed when modeling the $ReLU$ operator. (Note that minimizing the sum $x + s$ is not a viable option here, as this would alter the DNN nature and will tend to reduce the absolute value of the ReLU input).

To impose that at least one of the two terms x and s must be zero, one could add to (2) the bilinear (complementary) condition $x\,s \leq 0$, which is equivalent to $x\,s = 0$ as both terms in the product are required to be nonnegative.

A second option (which is the one we applied in our study) is to introduce a binary *activation* variable z and to impose the logical implications

$$\left.\begin{aligned} z = 1 \rightarrow x \leq 0 \\ z = 0 \rightarrow s \leq 0 \\ z \in \{0,1\} \end{aligned}\right\} \tag{3}$$

The above "indicator constraints" are accepted as such by modern MILP solvers, and are internally converted into proper linear inequalities of the type $x \leq M^+(1-z)$ and $s \leq M^- z$ (assuming that finite nonnegative values M^+ and M^- can be computed such that $-M^- \leq w^T y + b \leq M^+$) and/or are handed implicitly by the solution algorithm.

Using a binary activation variable z_j^k for each UNIT(j,k) then leads to the following 0-1 MILP formulation of the DNN:

$$\min \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=1}^{K} \sum_{j=1}^{n_k} \gamma_j^k z_j^k \tag{4}$$

$$\left.\begin{array}{r} \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - s_j^k \\ x_j^k, s_j^k \geq 0 \\ z_j^k \in \{0,1\} \\ z_j^k = 1 \rightarrow x_j^k \leq 0 \\ z_j^k = 0 \rightarrow s_j^k \leq 0 \end{array}\right\} k = 1,\ldots,K,\ j = 1,\ldots,n_k \tag{5}$$

$$lb_j^0 \leq x_j^0 \leq ub_j^0, \qquad j = 1,\ldots,n_0 \tag{6}$$

$$\left.\begin{array}{l} lb_j^k \leq x_j^k \leq ub_j^k \\ \overline{lb}_j^k \leq s_j^k \leq \overline{ub}_j^k \end{array}\right\} k = 1,\ldots,K,\ j = 1,\ldots,n_k. \tag{7}$$

In the above formulation, all weights w_{ij}^{k-1} and biases b_j^k are given (constant) parameters; the same holds for the objective function costs c_j^k and γ_j^k, that can be defined according to the specific problem at hand. Some relevant cases will be addressed in the next section. Conditions (5) define the ReLU output for each unit, while (6–7) impose known lower and upper bounds on the x and s variables: for $k = 0$, these bounds apply to the DNN input values x_j^0 and depend on their physical meaning, while for $k \geq 1$ one has $lb_j^k = \overline{lb}_j^k = 0$ and ub_j^k, $\overline{ub}_j^k \in \Re_+ \cup \{+\infty\}$.

Besides ReLU activations, some modern DNNs such as Convolutional Neural Networks (CNNs) [3, 10] involve multi-input units that perform the following *average/max pooling* operations:

$$x = AvgPool(y_1,\ldots,y_t) = \frac{1}{t} \sum_{i=1}^{t} y_i \tag{8}$$

$$x = MaxPool(y_1,\ldots,y_t) = \max\{y_1,\ldots,y_t\}. \tag{9}$$

The first operation (8) is just linear and can trivially be incorporated in our MILP model, while (9) can be expressed by introducing a set of binary variables $z_1,\cdots,z_t$ (that represent arg max) along with the following constraints:

$$\sum_{i=1}^{t} z_i = 1 \tag{10}$$

$$\left.\begin{array}{r} x \geq y_i, \\ z_i = 1 \rightarrow x \leq y_i \\ z_i \in \{0,1\} \end{array}\right\} i = 1,\cdots,t \tag{11}$$

It should be noticed that indicator constraints such as those appearing in (3) or (11), as well their bilinear equivalent like $x_j^k\, s_j^k \leq 0$, tend to produce very hard mixed-integer instances that challenge the current state-of-the-art solvers. As a matter of fact, the evaluation of the practical feasibility of model (4–7) was one of the main motivations of our work.

Discussion Here are some comments about the 0-1 MILP model above.

1. If one fixes the input x^0 of the DNN (i.e., if one sets $lb_j^0 = ub_j^0$ for all $j = 1, \ldots, n_0$), then all the other x variables in the model are fixed to a unique possible value—the one that the DNN itself would compute by just applying (1) by increasing layers. As to the binary z variables, they are also defined uniquely, with the only possible "degenerate" exception of the binary variable z_j^k corresponding to a ReLU unit that receives a zero input, hence its actual value is immaterial.
2. Because of the above, the 0-1 MILP model (4–7) cannot be infeasible, and actually *any* (possibly random) input vector x^0 satisfying the bounds conditions (6) can easily be extended (in a unique way) to produce a feasible solution. (Note that the same does not hold if one randomly fixes the activation binary variables z_j^k.) This is in contrast with other 0-1 MILP models with indicator (or big-M) constraints, for which even finding a feasible solution is a hard task. In this view, powerful refinement heuristics such as *local branching* [6], *polishing* [12], or *proximity search* [7] can be used to improve a given (possibly random) solution. This is important in practice as it suggests a hybrid solution scheme (not investigated in the present paper) in which initial heuristic solutions are found through fast methods such as gradient descent, and then refined using MILP technology.
3. It is known [1] that, for 0-1 MILP models like (4–7), the definition of tight upper bounds for the continuous variables appearing in the indicator constraints plays a crucial role for the practical resolution of the model itself. Modern MILP solvers are able to automatically define reasonable such upper bounds, propagating the lower/upper bounds on the input layer 0 to the other ones. However, these upper bounds can be rather inaccurate. We found that a much better option, very much in the spirit of [1], is instead as follows: We scan the units by increasing layers $k = 1, \ldots, K$. For the current $\texttt{UNIT}(j, k)$, we remove from the model all the constraints (and variables) related to all the other units in the same layer or in the subsequent ones, and solve twice the resulting model: one to maximize x_j^k and the other to maximize s_j^k. The resulting optimal values (or their optimistic estimate returned by the MILP solver after a short time limit) can then be used to define a tight bound on the two variables x_j^k and s_j^k, respectively. Due to the acyclic nature of the DNN, the tightened bounds computed in one iteration can be used in all the subsequent ones, i.e., the method always solves a 0-1 MILP with very tight upper bounds on the continuous variables. Note that, for a given DNN with fixed weights/biases, the tightened bounds do not depend on the input variables x_j^0 but only on their *a priori* lower/upper bounds lb_j^0/ub_j^0. As a result, the final tightened bounds can be saved in a file and reused for any future optimization of the same DNN. Note that the upper bound computation for each layer can be distributed (without communication) on a cluster of parallel computers, thus reducing preprocessing time.

3 Applications

Model (4–7) is (un)fortunately *not* suited for training. In training, indeed, one has a number of training examples, each associated with a different input x^0. So, in this setting, x^0 can be considered to be given, while the variables to optimize are the weights w_j^k and biases b_j^k. It then follows that, for any internal layer $k \geq 2$, the x's still play the role of variables (as they depend on the weights/biases of the previous layer), so the terms $w_{ij}^{k-1} x_i^{k-1}$ are in fact bilinear.

On the other hand, previous experiences of using MILP technology for training (such as the one reported in [5]) seem to indicate that, due to overfitting, insisting on finding proven optimal solutions is not at all a clever policy for training.

Instead, our model is more suited for applications where the weights are fixed, and one wants to compute the best-possible input example according to some linear objective function, as in the relevant cases discussed below. In those cases, indeed, overfitting the given DNN is actually a desired property.

3.1 Experimental setup

We considered a very standard classification problem: hand-written digit recognition of an input 28 x 28 figure. Each of the 784 pixels of the figure is normalized to become a gray-level in [0, 1], where 1 means white and 0 black. As in [13], the MNIST [3] dataset was used to train a simple DNN with 3 hidden layers with (8, 8, 8) ReLUs, plus a last layer made up of 10 units to classify digits "0" to "9", reaching (after 50 epochs) an accuracy of 93.04% on the test set.

3.2 Feature visualization

Following [4] we used our 0-1 MILP model to find input examples that maximize the activation x_j^k of each unit $\texttt{UNIT}(j, k)$. For our simple DNN, each of the resulting models could be solved within 1 second (and very often much faster) when using, on the fly, the bound strengthening procedure described in the previous section (the solver was more than one order of magnitude slower without this feature). Some max-activating input examples are depicted in Fig. 1, and show that no nice visual pattern can be recognized (at least, for our DNN).

It should be noticed that, before our work, the computation of the max activations was performed in the literature by using a greedy ascent method that can be trapped by local optimal solutions. According to our experience with a preliminary implementation of a gradient-based method, many times one is even unable to find any meaningful solution with

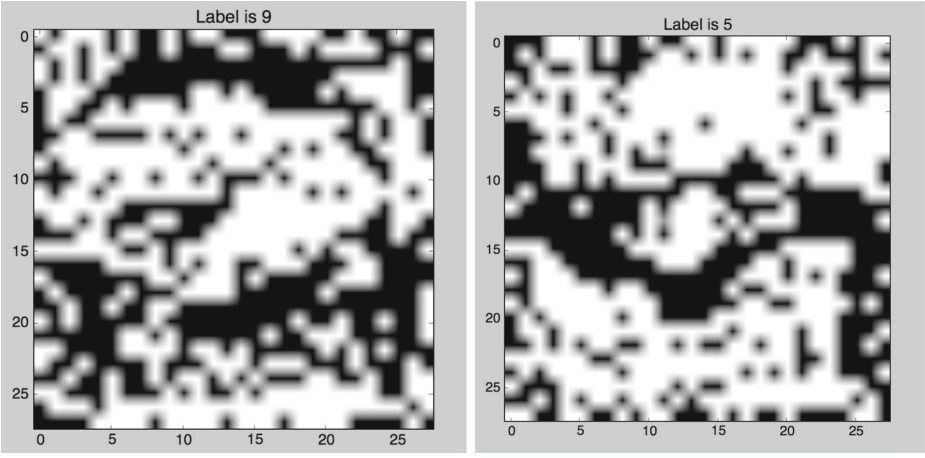

Fig. 1 Input examples maximizing the activation of some hidden units; no visual pattern can be identified in these provably optimal solutions

activation strictly larger than zero. In our view, the capability of finding provable optimal solutions (and, in any case, very good solutions)is definitely a strength of our approach.

3.3 Building adversarial examples

In our view, this is the most intriguing application of the MILP technology, due to its ability to compute (almost) optimal solutions that are intended to point out some hidden weaknesses of the DNN of interest. The problem here is to slightly modify a given DNN input so that to produce a wrong output. The construction of these optimized "adversarial examples" is the core of adversarial machine learning [14].

In our setting, we are given an input figure $\tilde{x}^0$ that is classified correctly by our DNN as a certain digit $\tilde{d}$ (say), and we want to produce a similar figure x^0 that is wrongly classified as $d \neq \tilde{d}$. We assume to play the role of a malicious attacker who has a complete access to the DNN (i.e., he/she knows its structure and parameters) and is allowed to arbitrarily change every single pixel of the input figure. We therefore do not impose any *a priory* perturbation pattern like blurring as it is done, e.g., in [15].

To show the flexibility of the MILP approach, in our experiments we also impose the actual (wrong) digit d that we want to obtain, by setting $d = (\tilde{d} + 5) \mod 10$. E.g., we require that a "0" must be classified as "5", and a "6" as a "1"; see Fig. 2 for an illustration. To this end we impose, in the final layer, that the activation of the required (wrong) digit is at least 20% larger than any other activations. Due to the MILP flexibility, this just requires adding to (4–7) the linear conditions

$$x^K_{d+1} \geq 1.2\, x^K_{j+1}, \quad j \in \{0, \ldots, 9\} \setminus \{d\}. \tag{12}$$

In order to reduce the L_1-norm distance between x^0 and $\tilde{x}^0$, we minimize the ad-hoc objective function $\sum_{j=1}^{n_0} d_j$, where the additional continuous variables d_j must satisfy the following linear inequalities to be added to model (4–7):

$$-d_j \leq x^0_j - \tilde{x^0_j} \leq d_j,\ d_j \geq 0, \qquad \text{for } j = 1, \ldots, n_0. \tag{13}$$

Figure 2 illustrates the power of the approach, in that the model is able to locate 2-3 critical pixels whose change tricks the DNN.

A key advantage of our method over previous approaches is that one can easily impose constraints like the one requiring that the final activation of the wrong label is at least 20% larger than any other activation. Similar constraints can be imposed to the input figure x^0, requiring e.g. a maximum number of changed pixels in the input figure, or a maximum deviation of each pixel with respect to the $\tilde{x}^0$; see Fig. 3 for an illustration. This control is an important novelty with respect to other adversarial example generation methods, and it can hopefully allow for a more qualitative probing of what exactly a DNN has learned. In addition, as Fig. 4 clearly shows, the optimized solutions are very different form the random noise used by the standard methods typically used in this kind of experiments.

4 Computational performance

This section is aimed at investigating the practical performance of a state-of-the-art MILP solver (`IBM ILOG CPLEX 12.7` in our case) to construct adversarial examples for not-too-small DNNs. We used the same experimental MNIST setup as in the previous section, and addressed DNNs with the following structure:

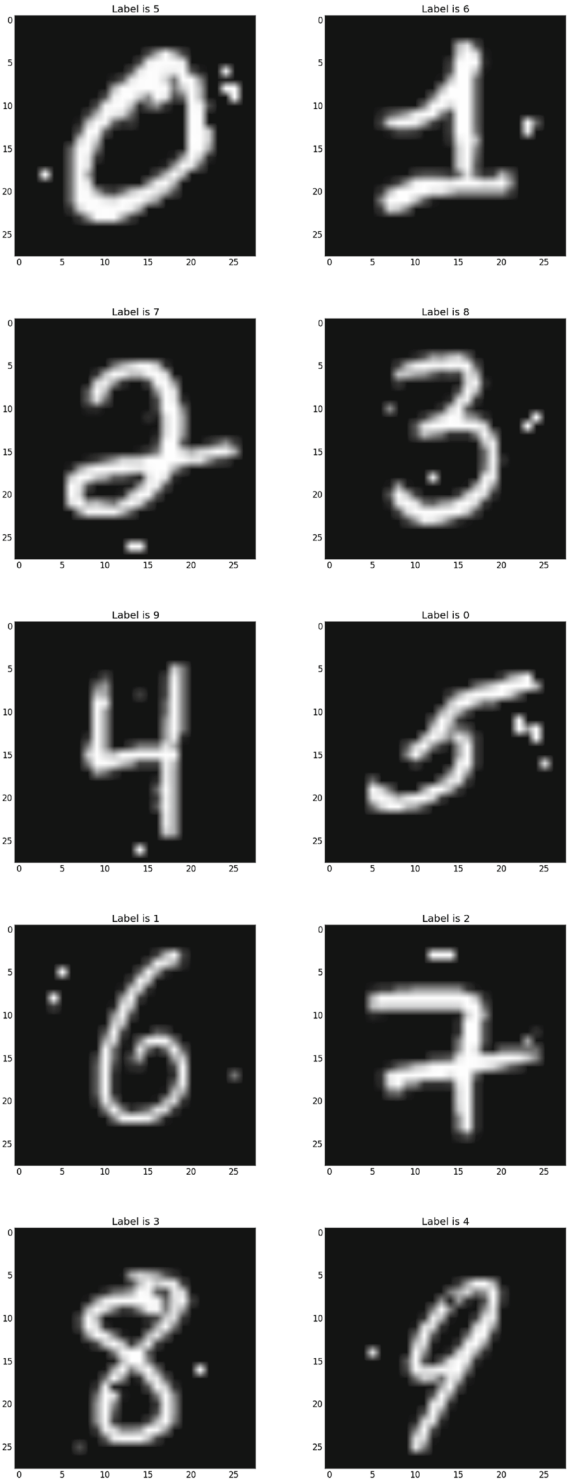

Fig. 2 Adversarial examples computed through our 0-1 MILP model; the reported label is the one having maximum activation according to the DNN (that we imposed to be the true label plus 5, modulo 10). Note that the change of just few well-chosen pixels often suffices to fool the DNN and to produce a wrong classification

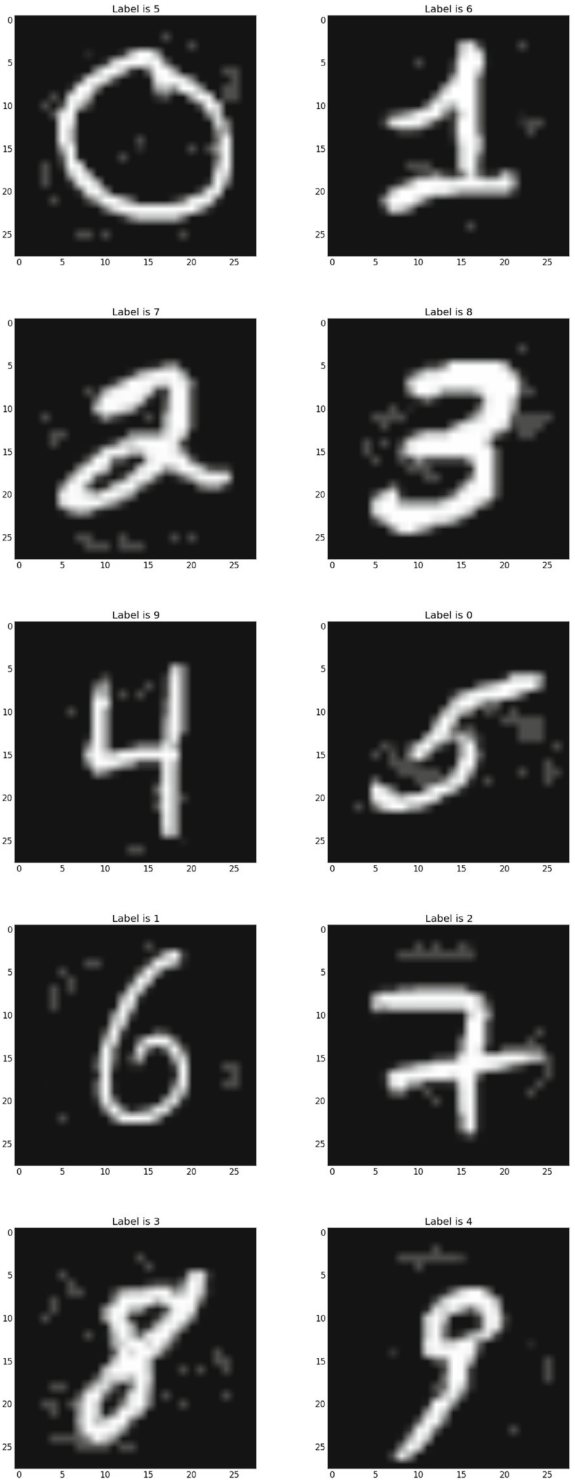

Fig. 3 Adversarial examples computed through our 0-1 MILP model as in Fig. 2, but imposing that the no pixel can be changed by more than 0.2 (through the additional conditions $d_j \leq 0.2$ for all j)

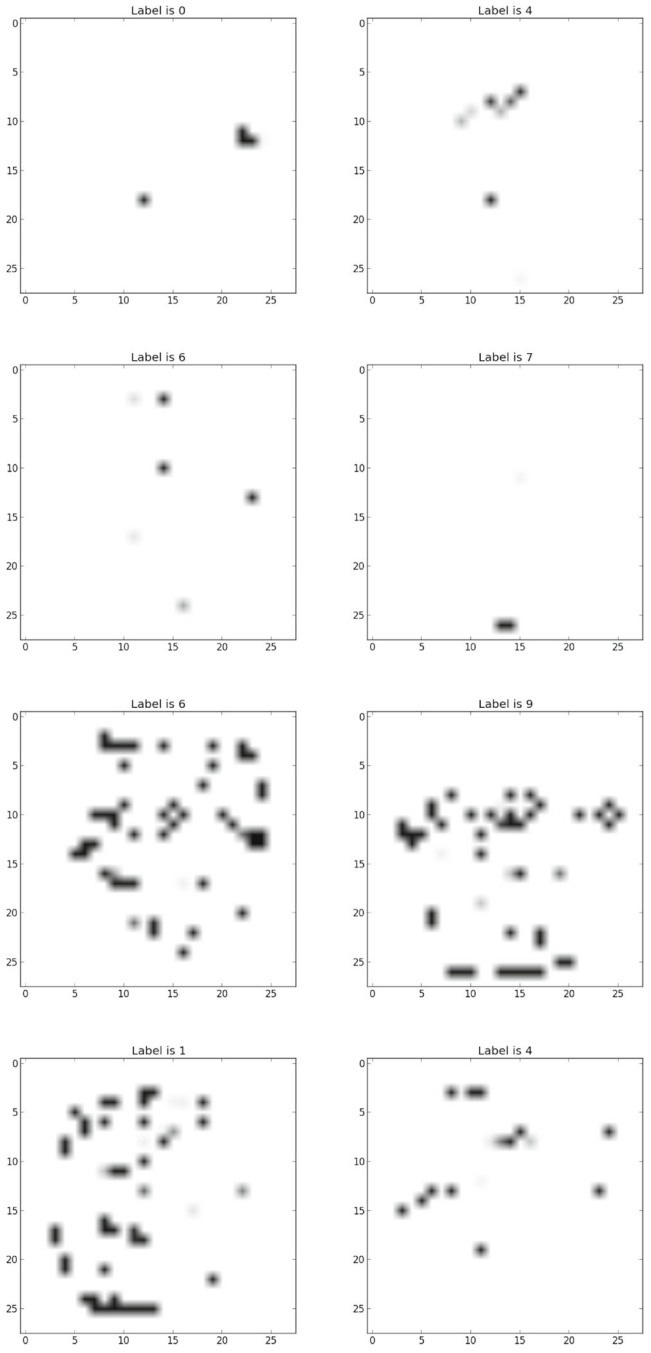

Fig. 4 Pixel changes (absolute value) that suffice to trick the DNN: the four top subfigures correspond to the model where pixels can change arbitrarily, while those on the bottom refer to the case where each pixel cannot change by more than 0.2 (hence more pixels need be changed). To improve readability, the black/white map has been reverted and scaled, i.e., white corresponds to unchanged pixels ($d_j = 0$) while black corresponds to the maximum allowed change ($d_j = 1$ for the four top figures, $d_j = 0.2$ for the four bottom ones)

– DNN1 : 8+8+8 internal units in 3 hidden layers, as in [13];
– DNN2 : 8+8+8+8+8+8 internal units in 6 hidden layers;
– DNN3 : 20+10+8+8 internal units in 4 hidden layers;
– DNN4 : 20+10+8+8+8 internal units in 5 hidden layers;
– DNN5 : 20+20+10+10+10 internal units in 5 hidden layers.

All DNNs involve an additional input layer (i.e., layer 0) with 784 entries for the pixels of the 28x28 input figure, and an additional output layer (i.e., layer K) with 10 units to classify the ten digits.

All DNNs were trained for 50 epochs using Stochastic Gradient Descent (SGD) and produced a test-set (top-1) accuracy of 93-96%. The best weights/biases were used to build our basic model (4–7), that was then modified for the adversarial case by adding the distance variables d_j's and the associated constraints (12–13). All d_j variables have an infinite upper bound, meaning that we do not impose any limit to the change of the input pixels; see Fig. 2 for an illustration of the typical adversarial examples computed through this model.

Before running the final experiments, the preprocessing phase described in Section 2 (item 3) was applied (for each DNN) to tighten the variable bounds. The tightened bounds for all the x_j^k and s_j^k variables were saved in a file and used in the runs reported in the present section under the label "improved model".

Table 1 reports some statistics of our runs (average values over 100 runs for each DNN and each model). Each run addressed the modification of a different MNIST training point, i.e., for each DNN and for each model we considered 100 different instances of the adversarial problem. Computational times refer to the use of the state-of-the-art MILP solver `IBM ILOG CPLEX 12.7` [9] on a standard 4-core notebook equipped with an Intel i7 @ 2.3GHz processor and 16 GB RAM—the GPU being not used by the MILP solver. A time limit of 300 seconds was imposed for each run (preprocessing time was not taken into account in these experiments).

In the table, column "%solved" reports the percentage of the instances that have been solved to proven optimality within the time limit, while columns "nodes" and "time (s)" give, respectively, the average number of branching nodes and of computing time (in wall-clock seconds) over all instances; time-limit instances count as 300 seconds. Finally, column "%gap" gives the percentage gap between the best upper bound and the best lower bound computed for the instance at hand (instances solved to proven optimality having a gap of zero).

Table 1 Comparison of the basic and improved models with a time limit of 300 seconds, clearly showing the importance of bound tightening in the improved model

	Basic model				Improved model			
	%solved	%gap	Nodes	Time (s)	%solved	%gap	Nodes	Time (s)
DNN1	100	0.0	1,903	1.0	100	0.0	552	0.6
DNN2	97	0.2	77,878	48.2	100	0.0	11,851	7.5
DNN3	64	11.6	228,632	158.5	100	0.0	20,309	12.1
DNN4	24	38.1	282,694	263.0	98	0.7	68,563	43.9
DNN5	7	71.8	193,725	290.9	67	11.4	76,714	171.1

In this experiment, the preprocessing time needed to optimally compute the tightened bounds is not taken into account

According to the table, the basic model gets into trouble for our three largest DNNs, as it was not able to solve to proven optimality a large percentage of the instances and returned a significant gap in the end. On the other hand, the improved model consistently outperforms the basic one, and starts having difficulties only with our most-difficult network (DNN5). The difference in terms of computing time and number of branching nodes is also striking.

In the previous experiment we concentrated on the computing time spent for the solution of a specific instance of the adversarial problem, without taking into account the preprocessing phase needed to optimally compute the tightened bounds. As a matter of fact, this preprocessing phase took negligible computing time for the three smallest DNNs, while for DNN4 (resp. DNN5) it took 1,112 (resp. 4,913) seconds in total, the most time consuming iteration requiring 151 (resp. 726) seconds. As the preprocessing is applied only once for each DNN, computing times of this order of magnitude can be considered acceptable in some applications. If this is not the case, as already mentioned, one could compute the bounds of each layer in parallel, and/or impose a short time limit of few seconds for each bound computation. Needless to say, the latter option (although mathematically correct) can reduce the tightness of some of the computed bounds, hence one is interested in evaluating the impact of the weaker bounds in the solution times of the single adversarial problems. To this end, in Table 2 we compare the performance of the improved model with exact bounds (as in Table 1) and with the weaker bounds obtained by imposing a very tight time limit of 1 sec. for each bound computation. (Almost identical results, both in terms of preprocessing time and of the effect on the improved model, have been obtained by aborting the MILP solver right after the root node.) The outcome of this experiment is quite encouraging, in that it shows that a fast preprocessing phase suffices in computing good bounds that significantly help the solution of the adversarial instances.

Table 3 analyzes the performance of the basic and improved models when one does not insist in finding a provable optimal solution, but content herself with a solution which is guaranteed to be within 1% from optimality. To be specific, in this experiment each run was interrupted as soon as the gap between the value of the best-known solution and of the best-known lower bound falls below 1%, meaning that the best-known solution is guaranteed to be, at most, 1% worse than the optimal one. As in Table 2, for the improved models we used (in the preprocessing phase) the weaker bounds computed with a time limit of 1 sec. for each bound computation. To limit the overall computing time for this experiment, a time limit of 3,600 seconds was imposed for each run, and the number of times this limit was reached (out of 100) is reported in the table in column #timlim. The table also reports the average number of nodes (nodes) and the average percentage gap (%gap), out of 100 runs;

Table 2 Performance of the improved model with a time limit of 300 seconds, with exact vs weaker bounds (the latter being computed with a time limit of 1 sec. for each bound computation)

	Improved model									
	Exact bounds					Weaker bounds				
	t.pre.	%sol.	%gap	Nodes	Time (s)	t.pre.	%sol.	%gap	Nodes	Time (s)
DNN4	1,112.1	98	0.7	68,563	43.9	69.4	98	0.4	80,180	45.5
DNN5	4,913.1	67	11.4	76,714	171.1	72.6	57	16.9	84,328	185.0

The overall preprocessing time (t.pre.) is greatly reduced in case of weaker bounds, without deteriorating too much the performance of the model. The difference w.r.t. the basic model in Table 1 is still striking

Table 3 Performance of the basic and improved model (the latter with the 1-sec. weaker bounds as in Table 2) to get solutions with guaranteed error of 1% or less; each run had a time limit of 3,600 seconds; the number of time limits, out of 100, is reported in column #timlim

	Basic model				Improved model (weaker bounds)			
	#timlim	Time (s)	Nodes	%gap	#timlim	Time (s)	Nodes	%gap
DNN1	0	1.0	1,920	0.5	0	0.6	531	0.3
DNN2	0	47.0	76,286	0.9	0	7.5	12,110	0.8
DNN3	8	632.8	568,579	2.2	0	11.3	19,663	0.9
DNN4	36	1806.8	1,253,415	10.2	0	50.0	89,380	1.0
DNN5	81	3224.0	1,587,892	43.5	11	851.0	163,135	3.8

averages also take time-limit instances into account. Note that the average gap is sometimes larger than 1%, due to the time limit imposed. The results show that the improved model—even with weaker bounds—greatly outperforms the basic model in this setting as well, thus confirming the results of Table 1. For all the considered DDNs but the largest one (DNN5), the MILP solver applied to the improved model was able to compute the required almost-optimal solutions (less than 1% from optimality) in a matter of seconds. As to DNN5, the improved model hit the time limit only 11 out of 100 times (while the basic model reached it 81 times); for the remaining 89 cases, the improved model required 511 seconds, on average, to reach 1% optimality.

5 Conclusions and future work

We have addressed a 0-1 Mixed-Integer Linear model for Deep Neural Networks with ReLUs and max/average pooling. This is a very first step in the direction of using discrete optimization as a core tool in the study of neural networks.

We have discussed the specificities of the proposed model, and we have described an effective bound-tightening technique to significantly reduce solution times. Although the model is not suited for training (as it becomes bilinear in this setting), it can be useful to construct optimized input examples for a given (already trained) neural network. In this spirit, we have reported its application to two relevant problems in Machine Learning such as feature visualization and adversarial machine learning. In particular, the latter qualifies as as a natural setting for mixed-integer optimization, in that one calls for (almost) optimal solutions that fool the neural network by "overfitting" it.

For small DNNs, our model can be solved to proven optimality in a matter of seconds on a standard notebook. However, for larger and more realistic DNNs the computing time can become too large. For example, even in the MNIST setting, DNNs of size (30, 20, 10, 10, 10, 8, 8, 8) or (50, 50, 50, 20, 8, 8) lead to computing times of one hour or more. In those hard cases, one should resort to heuristics possibly based on a restricted version of the model itself, in the vein of [6, 7, 12].

Future work should therefore address the reduction of the computational effort involved in the exact solution of the model, as well as new heuristic methods for building adversarial examples for large DNNs (possibly involving convolutional layers). Finding new deep learning applications of our mixed-integer model is also an interesting topic for future research.

Acknowledgements The research of the first author was partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-014, any by MiUR, Italy, through project PRIN2015 "Nonlinear and Combinatorial Aspects of Complex Networks". The research of the second author was funded by the Institute for Data Valorization (IVADO), Montreal. We thank Yoshua Bengio and Andrea Lodi for helpful discussions.

References

1. Belotti, P., Bonami, P., Fischetti, M., Lodi, A., Monaci, M., Nogales-Gomez, A., Salvagnin, D. (2016). On handling indicator constraints in mixed integer programming. *Computational Optimization and Applications*, *65*, 545–566.
2. Cheng, C.-H., Nührenberg, G., Ruess, H. (2017). Maximum resilience of artificial neural networks. In D'Souza, D., & Narayan Kumar, K. (Eds.) *Automated technology for verification and analysis* (pp. 251–268). Cham: Springer International Publishing.
3. Le Cun, Y.L., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of IEEE*, *86*(11), 2278–2324.
4. Erhan, D., Bengio, Y., Courville, A., Vincent, P. (2009). Visualizing higher-layer features of a deep network.
5. Fischetti, M. (2016). Fast training of support vector machines with Gaussian kernel. *Discrete Optimization*, *22*(Part A), 183–194. SI:ISCO 2014.
6. Fischetti, M., & Lodi, A. (2003). Local branching. *Mathematical Programming*, *98*(1-3), 23–47.
7. Fischetti, M., & Monaci, M. (2014). Proximity search for 0-1 mixed-integer convex programming. *Journal of Heuristics*, *20*(6), 709–731.
8. Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.
9. ILOG IBM. Cplex 12.7 user's manual (2017).
10. Krizhevsky, A., Sutskever, I., Hinton, G.E. (2017). Imagenet classification with deep convolutional neural networks. *Communication of ACM*, *60*(6), 84–90.
11. Nair, V., & Hinton, G.E. (2010). Rectified linear units improve restricted Boltzmann machines. In Fürnkranz, J., & Joachims, T. (Eds.) *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 807–814): Omnipress.
12. Rothberg, E. (2007). An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, *19*(4), 534–541.
13. Serra, T., Tjandraatmadja, C., Ramalingam, S. (2017). Bounding and counting linear regions of deep neural networks. CoRR arXiv:1711.02114.
14. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R. (2013). Intriguing properties of neural networks. CoRR arXiv:1312.6199.
15. Tjeng, V., & Tedrake, R. (2017). Verifying neural networks with mixed integer programming. CoRR arXiv:1711.07356.