Thesis in Quantitative Logistics

# Constructing Adversarial Examples for Deep Neural Networks using Mixed Integer Linear Optimization and Local Search Heuristics


ERASMUS UNIVERSITY ROTTERDAM

Erasmus School of Economics

Author

Leoni van Dijk

Supervisor

MSc. B.T.C. van Rossum

Second assessor

Dr. T.A.B. Dollevoet


July 4, 2021


**Abstract**

Deep Neural Networks (DNNs) are a popular type of machine learning architecture, due to their ability to learn from experience. A common weakness of DNNs is their likeliness to overfit a model, causing their performances to be inaccurate in practise. In this paper, we use a 0-1 Mixed Integer Linear Programming (MILP) formulation to model given DNNs, following the research of Fischetti and Jo (2018). We show that this formulation can be used to gain insights into features of the network and to investigate the network's accuracy. For the latter, we use the MILP to construct (optimal) adversarial examples capable of 'fooling' the network such that they are falsely classified. As our results show that this process is computationally costly for larger DNNs, we propose a general framework for a local search matheuristic. We experiment on multiple interpretations of this heuristic and show that it is capable of approaching optimal solutions for instances that generate favorable initial solutions.

# Contents

# 1 Introduction

*Deep Neural Networks* (DNNs) are a popular type of machine learning architecture, due to their ability to learn from experience (Goodfellow, Bengio, Courville, & Bengio, 2016). A DNN is built up out of multiple layers, each consisting of a certain number of neurons. This structure allows the network to partition complicated inputs into separated characteristics. During the training process, the DNN learns in what way certain combinations of characteristics are related to classify inputs correctly.

For the first layer of the DNN, inputs must be translated to specific values for the input neurons (values which will be further referred to as *activations*). For example, for DNNs built to recognize hand written digits such as in Erhan, Bengio, Courville, and Vincent (2009), every neuron of the input layer corresponds to a pixel of the image, where the activation of the neuron represents the pixel's opacity. In *fully connected layers*, the activation of neurons in any *hidden layer* (i.e., not an input layer) or output layer is obtained by performing arithmetic operations on all activations of the previous layer. Additionally, a nonlinear operator is applied to every activation, such as the commonly used *rectified linear unit* (ReLU) (Nair & Hinton, 2010). This operator returns the maximum between zero and the activation input.

Now that we have introduced in what way the different layers in DNNs are connected, one can understand that DNNs can be treated as piecewise linear functions (Serra, Tjandraatmadja, & Ramalingam, 2018). For a given network with known parameters, feeding the DNN input will result in a certain output. This enables us to model a DNN as a *Mixed Integer Linear Program* (MILP), as is done by Fischetti and Jo (2018). Such formulations can then be used to evaluate various properties of the DNN, specified by some objective function. In the past four years, applications of this MILP approach for DNNs have been appearing in scientific literature, together with techniques on how these problems can be solved more efficiently. Despite these developments, solving MILPs for larger DNNs seems to remain a huge challenge.

The first goal of this research is to replicate and validate the results of Fischetti and Jo (2018). In their paper, characteristics of five DNNs of increasing complexity are investigated by means of a 0-1 MILP formulation. First, the formulation is used to construct input examples that maximize the activations of specific neurons. Then, another model is solved to construct so-called *adversarial examples*. Adversarial examples are input examples that the DNN was able to correctly classify, but that are now slightly perturbed such that it fools the DNN. These examples are of great use to

evaluate the accuracy and robustness of the network. The DNNs used in their research all contain no more than 70 neurons in hidden layers, yet computational difficulties are experienced for the largest networks.

To extend the research of Fischetti and Jo (2018), the second goal of this research is to build a heuristic capable of constructing adversarial examples for larger DNNs within reasonable computational time. To our knowledge, heuristics for this purpose do not yet exist in the scientific literature. The construction of a heuristic is an iterative process of trial and error, for which we will set out the foundations in this paper. In its core, a heuristic consists out of three building blocks: a search rule, a stopping rule and a decision rule (Gigerenzer, 2008). We will perform various experiments for different configurations of these building blocks in the search of a heuristic of good quality, hopefully while benefiting from the MILP formulation.

The remainder of this paper is organized as follows. Section 2 will give an overview of the scientific literature concerning our problem. The methodology used for the replication and extension is stated in section 3, which is followed by an overview of our data and experimental setups in section 4. The results of the experiments will be presented and discussed in section 5, followed by a conclusion with recommendations for future research in section 6.

## 2 Literature review

Modelling DNNs using the 0-1 MILP framework is a relatively recent development in Artificial-Intelligence centered literature. The contribution of the MILP approach proposed in these articles is substantial. For a given - and thus already trained - DNN it can be used to investigate all sorts of strengths and weaknesses of the network. For instance, MILP formulations can be used to study the complexities of given DNNs (Serra et al., 2018), or to simplify them (Serra, Kumar, & Ramalingam, 2020).

A common weakness of DNNs is their likeliness to over fit a model (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). A DNN subject to overfitting is likely to be vulnerable to adversarial attacks (Szegedy et al., 2014). Recent literature reveals that the flexibility of the MILP approach is of great use to evaluate a DNN's vulnarability to such adversarial examples. Cheng, Nührenberg, and Ruess (2017) solve a MIP to compute maximum bounds on how much sensory input can be perturbed while still giving correct classifications. Fischetti and Jo (2018) use similar LP constraints in their formulation to show how MILP models can be used for constructing

adversarial examples. Additionally, they present a bound-tightening technique for the continuous variables in 0-1 MILP models to reduce solution times.

Yet still being connected to the evaluation of DNN accuracy, a slightly more general application of the MILP framework is *the reachability problem*. Reachability analysis aims to find out whether a certain state can be reached given the initial states of the system. Lomuscio and Maganti (2017) introduce the link between this problem and linear programming. Dutta, Jha, Sankaranarayanan, and Tiwari (2018) combine the use of local search with MILP solvers to estimate the range of possible output values the network given certain constraints on inputs. This seems to outperfom using solely a MILP approach for larger networks with over 1000 neurons, however it still needs excessive computational times for some of the larger networks.

Above mentioned articles are limited in the sense that their methods are only applicable to small networks. MILP solvers are not able to find solutions for larger DNNs, considering the combinatorial time and memory complexity these have. Cheng et al. (2017) propose a number heuristics for MIP to reduce solving times, but these are not tested for larger DNNs in their research. Tjeng, Xiao, and Tedrake (2019) were the first to verify adversarial test accuracy for DNNs with over 100.000 neurons. This is done by using a progressive bound tightening technique on the inputs to non-linearities and via a presolve algorithm that uses the information from a restricted input domain. However, much work still needs to be done in the search for faster optimization methods.

In this paper, we will follow the recommendations of Fischetti and Jo (2018) and try to find heuristic methods to construct adversarial examples for larger DNNs within reasonable computation time. Together, Wilbaut and Hanafi (2009) and Fischetti and Lodi (2010) give a good overview of the existing heuristic methods to solve 0-1 MILPs, of which the greatest part is based on a relaxation of the integer variables. However, modern MILP solvers make use of the well known big-M strategy to deal with indicator constraints as proposed in the formulation of Fischetti and Jo (2018), of which LP relaxations are weak (Belotti et al., 2016). As a result, the use of these methods does not seem promising for our problem. Park and Boyd (2017) and Gigerenzer (2008) suggest more general frameworks for (local-search) heuristics, on which we intend to base our novel algorithms.

As we are trying to construct adversarial examples as close to the original figure as possible, local search algorithms seem to be very suitable. A *matheuristic* is a heuristic solving method in which a mathematical programming model is used to solve smaller subproblems (Fischetti & Fischetti, 2018). We will try to construct a local search matheuristic as this allows us to benefit from the proposed MILP formulation for DNNs, provided in the first part of the paper.

# 3 Research design and Methods

In this section, the methods used in this paper will be discussed. The first two subsections give a summary of the methods used by Fischetti and Jo (2018), as the first goal of this research is to replicate and verify their results. Section 3.1 explains how DNNs with ReLUs can be formulated in the MILP framework. Furthermore, a bound thightening procedure will be explained which reduces solution times. Section 3.2 shows how this general MILP formulation can be extended to solve two types of applications. Finally, in section 3.3 the research methods that we will use to construct the heuristic will be presented.

## 3.1 A 0-1 MILP Model

As is briefly mentioned in the introduction, one can view a DNN with given parameters as a piecewise linear function. The activation of every neuron in the network is the result of a series of mathematical operations on the input values. In this section, the formulations used to model DNNs as a 1-0 MILP are presented, following the work of Fischetti and Jo (2018). This MILP formulation can then be used to find optimal examples of input figures for certain features of the network one wants to investigate.

### 3.1.1 DNN structure

A DNN is built up out of multiple layers, each consisting of a certain number of neurons. This structure is visualized by Figure 1, which gives an illustration of a simple neural network. Every neuron has its own activation value, which is the result of a weighted sum of the activations of the neurons in the previous layer, adding a bias and the performance of a non-linear operator.

The following notation is used. Every layer of the DNN is numbered from 0 to $K$, resulting in a total of $K+1$ layers. Layers 0 and $K$ correspond to the input- and output layers respectively. Every layer $k$ consists out of $n_k$ neurons (or *units*), each having an activation value $a^k_{n_i}$. Let $a^k$ be the complete activation vector of layer $k$. Generally, for any $k \geq 1$, the activations $a^k$ are computed through the following formula:

$$a^k = \delta(W^{k-1}a^{k-1} + b^{k-1}) \tag{1}$$

where $\delta(.)$ represents a non-linear function, for which in this paper is chosen for the ReLU. Moreover, $W^{k-1}$ is a matrix of weights and $b^{k-1}$ is a vector of biases, both containing trainable parameters.

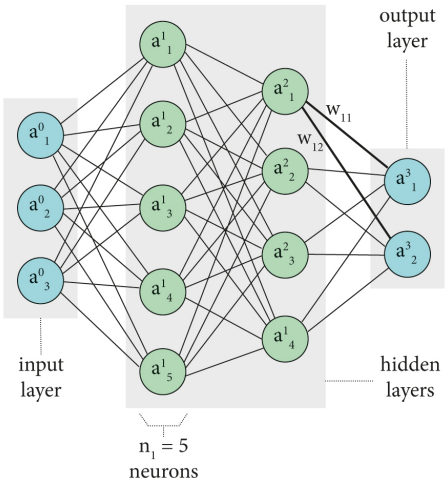



**Figure 1** Simple example of a neural network with 5+4 internal units in 2 hidden layers.

### 3.1.2 Encoding ReLU

The so-called *rectified linear unit* (ReLU) (Nair & Hinton, 2010) is a commonly used non-linear activation function in DNNs. Avoiding negative activations, its output value is just the maximum between its input value and zero. To be able to translate this into a MILP model, one needs to consider the following equations.

$$a = ReLU(w^T y + b) = \begin{cases} w^T y + b & \text{if } w^T y + b \geq 0 \\ 0 = w^T y + b + s & \text{if } w^T y + b < 0, \quad s \geq 0 \end{cases} \tag{2}$$

which can be summarized into the linear conditions

$$w^T y + b = a - s \quad s > 0, \quad a \geq 0 \tag{3}$$

This constraint on its own is not sufficient in modelling the ReLU operator. To ensure the uniqueness of solutions $(x, s)$, a binary activation variable $z$ is introduced to impose a set of indicator constraints. These constraints are accepted by the solver as big-M constraints and are as follows.

$$\begin{cases} z = 1 & \rightarrow \quad a \leq 0 \\ z = 0 & \rightarrow \quad s \leq 0 \\ z \in \{0, 1\} \end{cases} \tag{4}$$

### 3.1.3 (basic) Model formulation

The basic structure of a MILP formulation for a DNN with fully connected ReLU layers can now be presented as the following.

$$\min \quad \sum_{k=0}^{K}\sum_{j=1}^{n_k} c_j^k a_j^k + \sum_{k=1}^{K}\sum_{j=1}^{n_k} \gamma_j^k z_j^k \tag{5}$$

$$\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} a_i^{k-1} + b_j^{k-1} = a_j^k - s_j^k \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{6}$$

$$z_j^k = 1 \rightarrow a_j^k \leq 0 \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{7}$$

$$z_j^k = 0 \rightarrow s_j^k \leq 0 \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{8}$$

$$lb_j^0 \leq a_j^0 \leq ub_j^0 \qquad j = 1,\dots,n_0 \tag{9}$$

$$lb_j^k \leq a_j^k \leq ub_j^k \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{10}$$

$$\bar{lb}_j^k \leq s_j^k \leq \bar{ub}_j^k \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{11}$$

$$a_j^k, s_j^k \geq 0 \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{12}$$

$$z_j^k \in \{0,1\} \qquad k = 1,\dots,K, j = 1,\dots,n_k \tag{13}$$

The objective function in equation (5) serves as an example and can be defined according to the specific application one is studying. Here, $c_j^k$ and $\gamma_j^k$ are constant parameters. As this model formulation is not suitable for training, weights and biases parameters $w_{ij}^k$ and $b_i^k$ are assumed to be known. Constraints (6)-(8) define the ReLU output as described in section 3.1.2, in combination with the bounds imposed on the variables in (12) and (13). Finally, constraints (9)-(11) allow for tighter bounds on these variables. In the next section, a method will be presented to tighten the bounds for our specific case.

### 3.1.4 Bound tightening procedure

The big-M strategy modern MILP solvers use when dealing with indicator constraints comes with the drawback of resulting in weak MILP relaxations (Belotti et al., 2016). Fischetti and Jo (2018) propose a bound-tightening mechanism to strengthen the weak bounds coming from these relaxations, and thus reduce the solution times. This mechanism contains features of the *Iterative domain reduction* technique proposed by Belotti et al. (2016), where a sequence of MIPs is solved.

The bound tightening procedure forms the preprocessing phase of solving the MILP. First, for a given DNN with weights and biases, lower and upper bounds must be chosen for the input layer.

These bounds depend on their physical meaning and highly influence the bounds of the activations in further layers. In our case, activations of the input layer must be between 0 and 1, as our input units represent grey-levels of pixels. This thus means that $lb_j^0 = 0$ and $ub_j^0, = 1$, for $j = 1, \dots n_0$. After this, for every neuron of every layer $k = 1, \dots K$, upper bounds on the $a_j^k$ and $s_j^k$ variables are calculated by solving two smaller MILP models. This can be done either exact or by using the slightly weaker bounds found by terminating the calculation after a short time limit. To find upper bounds on the variables of the current neuron, all constraints and variables related to all other neurons in the current layer and in the subsequent ones are removed. For this smaller model, two maximizations are performed; one for the value of $a_j^k$ and one for $s_j^k$. When solving these models for deeper layers of the DNN, the tightened bounds of the previous layers constrain the optimization.

After having calculated all bounds in every layer, they can be saved to use in any optimization for the same DNN.

### 3.2 Application models

In this subsection, the methods that Fischetti and Jo (2018) use to apply the MILP model for two applications are described. For the replications, the same dataset and DNN structures will be used as in this paper, which will be further elaborated on in the section 4. The first application concerns *feature visualization* (Erhan et al., 2009), where the MILP model is used to find input examples that maximize the activation any neuron j of layer k in the network. This will be done by maximizing $a_j^k$ as the objective function. Fischetti and Jo (2018) showed that no nice visual pattern could be recognized when investigating some max-activating input examples generated for a simple DNN. We will compare our results of this procedure to verify their conclusion.

For the second application, the model formulation is used to construct so-called *adversarial examples* (Szegedy et al., 2014). These examples are slightly perturbed inputs such that the DNN produces incorrect output, which are of great use to evaluate the network's accuracy. Formally, for a given input figure $\bar{a}^0$ which the DNN correctly classifies as being a certain digit $\bar{d}$, we now need to find a similar figure $a^0$ which is incorrectly labeled as $d \neq \bar{d}$. In the model is chosen for a predetermined misclassification digit, namely $d = (\bar{d} + 5) \bmod 10$. This means that a "0" image should be classified as "5", and a "6" as a "1". For this application, we add the following constraints to the general MILP formulation:

$$\min \quad \sum_{j=1}^{n_0} d_j \tag{14}$$

$$a_{d+1}^K \geq 1.2 a_{j+1}^K \qquad j \in \{0, \ldots, 9\} \setminus \{d\} \tag{15}$$

$$-d_j \leq a_j^0 - \bar{a}_j^0 \leq d_j \qquad j = 1, \ldots, n_0 \tag{16}$$

$$d_j \geq 0 \qquad j = 1, \ldots, n_0 \tag{17}$$

Where $d_j$ is a continuous variable minimizing the amount of perturbation that will be added by means of constraint (16) and objective function (14). Constraint (15) imposes that the activation of the required wrong digit is at least 20% higher than the activations of all other digits.

### 3.3 A local search matheuristic

When tackling models with excessive computation times, it is common to use heuristics to approximate optimal solutions. A great advantage of the 0-1 MILP framework for modelling DNNs is that it enables the use of matheuristics. In a matheuristic, the mathematical programming model is used to solve smaller subproblems which are used within the heuristic environment (Fischetti & Fischetti, 2018). In our case, we seek to find an adversarial example as close to the original input figure as possible. This objective suits the idea of performing a *local search*, where close neighbourhood solutions of an initial solution are investigated. In the remainder of this section, we will present the methods that we will use to place local search into the matheuristic framework.

Pseudocode 1.1 presents the general outline of the matheuristic, which we will further refer to as the *Cheapest Deceive Heuristic* (CDH). In this algorithm, perturbations are added iteratively until the stopping condition is reached. This way, no more perturbation is added than necessary for the DNN to reach sufficient confidence in the classification of the wrong label. In this code, `A`$^*$ stands for the highest activation in the output layer, without the activation of the `FalseLabel`. The activation for `FalseLabel` is represented by `FA`. In line 7 an objective is added to the model and in line 8 a decision rule is used. For the results to approximate optimal values we need to discover what are effective and efficient choices for these aspects of the heuristic. We outline some candidates for these choices that we will experiment on in the following subsection.

**Pseudocode 1.1** Cheapest Deceive Heuristic (CDH) - General Outline

```
Image ← correctly classified image,   Perturbations ← array of zeros,
Output ← compute output layer activations for current Image,
FalseLabel ← (correct label + 5) mod 10,   Ratio ← FA/A*
Search in the neighborhood of Image
while Ratio is not larger or equal than 1.2 do
    for every input neuron j in the neighborhood do
        initialize basic MILP model and tighten bounds
        initialize disturbance variables and add constraints (16),(17) to the MILP
        set UB = LB = Image[i] for every i is not j in the input layer
        add objective to model and solve
    perform perturbation according to decision rule:
        add value of d_j to Perturbations and Image, compute output activations
        Ratio ← FA/A*
```

### 3.3.1 Building blocks

In this subsection, we view the CDH as a modular system. Generally, a local search heuristic is built out of three building blocks (Gigerenzer, 2008). First we have the search rule, determining what the searching area is and how it is going to be investigated. Then, a decision rule specifies which solution is accepted for the current iteration. Iterations are performed until we reach the stopping rule, which in our case is reaching the ratio of 1.2. On top of the three general building blocks, we consider some solve rules. These rules involve procedures to avoid local optima and different methods to solve the subproblems. In the remainder of this subsection, we discuss several interpretations of these building blocks, of which we will evaluate the performances in the results section.

#### Search rules

The search rules for this heuristic can be divided into rules regarding the size of the neighbourhood and the objective functions used to inspect this neighbourhood. For our experiments we use two types of objective functions:

1. *Simple objective*: $max \quad a_j^K$,
2. *Diff objective*: $max \quad a_j^K - \sum_{i=1, i\neq j}^{10} a_i^K$ where j is the desired false label.

The first objective function maximizes the activation of the neuron corresponding to the desired false label. This is simple in the sense that the solver is only focused on maximizing the

value of one specific variable. The second objective tries to find a solution that maximizes the difference between the activation of the false label-neuron and those of the other output layer neurons. This increases the complexity of solving the subproblems, but could decrease the number of iterations necessary to reach stopping rule due to a more directed search.

As stated in Pseudocode 1.1, for every neuron in the searching area we construct and solve a MILP. This causes the algorithm to require more computational time if the searching area is large. For the search rules regarding the size of the neighborhood we experiment with the following:

1. *Complete neighborhood*
2. *Constrained neighborhood*

When we speak of searching in the complete neighborhood, we refer to the process of iterating over all 784 input neurons. To reduce the number of calculations performed during an iteration, we propose a heuristic approach that focuses on a smaller area of the input image, which we call the constrained neighborhood. This process is based on the idea of Xue, Yuan, He, Wang, and Liu (2021), who use an adaptive mask to constrain the area and intensities of added perturbations. For this, we construct a *probability mask* based on the set of images the DNNs are trained on. This probability mask constrains the searching area to neurons which have a probability larger than 0.01 to be activated if it was part of an image portraying the label, plus the neurons that have an activation value larger than zero in the current input figure. The probability mask is further elaborated on in section 4.

**Solve rules**

When constructing a matheuristic, one needs to make decisions on the size of the subproblems that are being solved. We expect that the bigger the subproblems are, the better the matheuristic is able to approach a solution computed by an exact algorithm. However, bigger subproblems come with the drawback of requiring more computational time. We experiment on this trade-off by means of the following subproblems:

1. *Unbounded perturbation*
2. *Fixed absolute perturbation*

For the first subproblem, the solver computes the best feasible perturbation value of an individual neuron in the input layer corresponding to the objective value. For the second subproblem, the solver only needs to make the decision of adding or subtracting a certain fixed parameter.

If both of these options are not feasibily possible (due to constraints on the bounds of the input activations), the operation is performed for which the perturbation parameter needs to be decreased the least. For this second subproblem, we expect to find a trade-off in computional time and objective quality for different values of the perturbation parameter, as low values require more iterations but approach the required ratio with more precision.

During every iteration of the heuristic, the solution that is accepted according to the decision rule highly depends on the current image that is being perturbed. If we reach a configuration for which the algorithm does not see any possibility of increasing the current ratio, we have reached a local optimum. In their description of the framework and applications of iterated local search algorithms, Lourenço, Martin, and Stützle (2019) state that a random move in the neighborhood of a solution can often result in an escape from the local optimum. Therefore, in an attempt to escape from these points, we consider the following methods:

1. *Random perturbation*
2. *Guided random perturbation*

For the first method, we draw a random neuron from the input layer and evaluate its activation. If it is lower or than 0.5, we set its value to 1 and if it is higher than 0.5 we set it to zero. For the second method, we make use of the probabilities computed for the constrained neighborhood search rule. If not yet fully activated, we set the activation of a randomly selected input neuron with an activation probability larger than $p^h$ to 1. Moreover, if not yet fully deactivated, we set the activation of a randomly selected input neuron with an activation probability lower than $p^l$ to 0. This way we hope to guide the randomization method towards perturbations that are likely to help increasing the ratio.

The randomization methods contain several parameters. $maxRand$ indicates the maximum number of times a randomization may be performed for a single instance. $k$ represents the number of random neurons that are perturbed at once. We expect that higher values for $k$ lead to a faster escape of the local point, but also to higher objective values as these perturbations are non-controlled. For a complete list of parameters and their default values in our experiments we refer to appendix B.

**Decision rules**

The decision rule specifies what aspects of a solution we consider to be 'good' or 'promising' to reach the stopping rule as efficiently as possible. In this research we consider the following two possibilities:

1. *Largest increase in ratio*
2. *Largest increase in objective value*

For the sake of time and the length of this paper, for every experimental configuration of the heuristic we choose the biggest increase in confidence ratio as the decision rule. Small experiments were performed for the second rule, but we decided to stop investigating this option as we noticed that this process was much more prone to get stuck in local optima and required more iterations.

## 4 Data and Experimental setup

To replicate the results of Fischetti and Jo (2018) we use five DNNs with different structures. Each DNN is trained to recognise handwritten digits by using the MNIST database (LeCun & Cortes, 2010). This database contains 60.000 training examples and 10.000 test examples. Every example consists of the correct label and a 28 x 28 image, for which each of the 784 pixels holds a value between 0 and 1. These values indicate the grey-levels of the figure where 0 means white and 1 means black. For the probability mask used in the heuristic experiments we performed a small probability analysis of the different images present in the MNIST training set. To see which input neurons are typically activated for a specific label, we summed all possible images of this label and then divided all activations by the number of images present for this label. Visual results of this process can be found in Appendix A. Note that this procedure is promising for our specific training set, but not so much for training sets where for example the digits are not centered.

Table 1 gives an overview of the characteristics of the five DNNs we use in our experiments. The structure of a DNN is described by a sequence of integers. Every integer represents a layer in the network, its value denoting the number of neurons within this layer. Hidden layers are made bold.

**Table 1** Characteristics of the DNNs used

| Model | Structure | Hidden layers | Trainable parameters | Test-set accuracy |
|---|---|---|---|---|
| `DNN1` | 784 - **8 - 8 - 8** - 10 | 3 | 6,280 | 0.93 |
| `DNN2` | 784 - **8 - 8 - 8 - 8 - 8 - 8** - 10 | 6 | 6,730 | 0.91 |
| `DNN3` | 784 - **20 - 10 - 8 - 8** - 10 | 4 | 16,160 | 0.96 |
| `DNN4` | 784 - **20 - 10 - 8 - 8 - 8** - 10 | 5 | 16,232 | 0.95 |
| `DNN5` | 784 - **20 - 20 - 10 - 10 - 10** - 10 | 5 | 16,660 | 0.96 |

Every DNN is trained in the same setting as in Fischetti and Jo (2018), meaning that they are trained

for 50 epochs and *Stachastic Gradient Descent* (SGD) is used for optimization. For training, we use the Python programming language (Van Rossum & Drake Jr, 1995) and especially the Keras package for deep learning (Chollet et al., 2015). While training, it turns out that using ReLU as the nonlinear operator in every single layer does not result in the same high test-set accuracy as Fischetti and Jo (2018) report in their paper. This problem has been adressed to the authors but sadly a response has yet to be given. As it is important to have DNNs with a high test-set accuracy to replicate further results of Fischetti and Jo (2018), we solve this problem by using the *soft-max* operator in the output layers. This operator squeezes every input value of these neurons to a value between 0 and 1, and transforms the output vector such that its values sum up to 1. The use of this soft-max operator has no effect on the replication of the results by means of the MILP formulation. This is because the soft-max operator is merely a normalization, where the rankings between highest and lowest activations do not change.

# 5 Results

In this section, the computational results of the methods discussed in section 3 are presented. Computations are performed by the state-of-the-art MILP solver `IBM ILOG CPLEX 20.1` (IBM-Corporation, 2020) on a notebook with an Intel Core i5-8250U CPU at 1.60GHz and 8 GB RAM. Note that the notebook of Fischetti and Jo (2018) is equiped with 16 GB RAM, possibly causing differences in computational times. This section is devided into two parts. In the first subsection, we present replications of the results of Fischetti and Jo (2018) and compare our results with those stated in their paper. Then, this section continues by presenting the results of the experiments performed in the process of constructing a heuristic for adversarial examples.

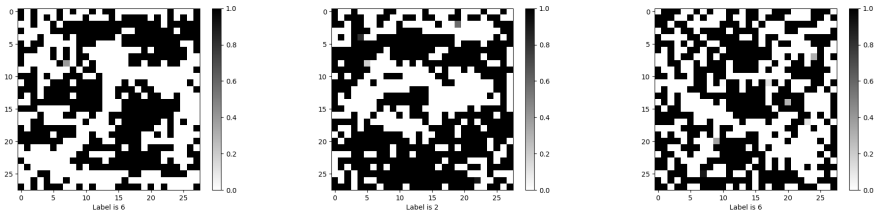
## 5.1 Replication

In fashion of the results provided by Fischetti and Jo (2018), in the following sections we make use of three types of models. The model as described in section 3.1.3 will be referred to as the "basic model". The "improved model" is the basic model incorporating the bounds calculated in the (exact) bound tightening procedure explained in section 3.1.4. Finally, when speaking of the "improved model with weaker bounds", we refer to the basic model for which the bounds are obtained by terminating each bound computation after 1 second. In the remainder of this subsection, we first perform a feature visualisation and then continue with the construction of adversarial examples.

### 5.1.1 Feature Visualisation

A common problem with the use of DNNs is that there exists no clear theoretical understanding of the way these networks learn. In the hope to gain some insight in the patterns a network possibly recognizes, one can perform a *Feature Visualisation*, as is done by Fischetti and Jo (2018). They show results for two hidden units and find no visual patterns. Figure 2 presents the results that we obtained when replicating their approach. Figures 2a and 2b resemble those of Fischetti and Jo (2018) but we do not agree that no visual pattern can be identified. Even though it is not very clear, one could argue that a slight curve is visible in 2a and a that the dark pixels in 2b are in a somewhat circular shape.

Another common issue with DNNs is the problem of *overfitting*. In this case the network fails to capture the generality of the inputs, causing it to perform well on training data but poorly on test data. This is a problem that gets more serious for larger networks (Hagiwara & Fukumizu, 2008). When comparing an optimal input example for `DNN4` (Figure 2c), which is a larger network, to the input examples for `DNN1` (Figures 2a, 2b), one can observe that the pixels in this image are even more scattered than those of the other two images. This result illustrates the concept of overfitting fairly well. For computations the improved model of `DNN4` was used with weak bounds, which was solved to optimality in a matter of seconds.

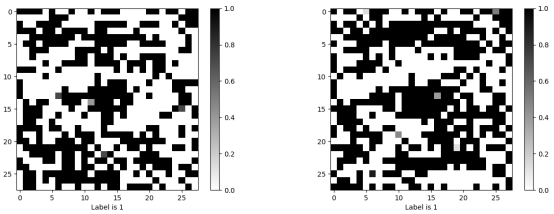



**(a)** Input for layer 2, neuron 2 (`DNN1`) **(b)** Input for layer 3, neuron 7 (`DNN1`) **(c)** Input for layer 4, neuron 5 (`DNN4`)

**Figure 2** Input examples that maximize the activations of given hidden units.

Finally, the feature visualization formulation can be used to investigate to what extent the network understands what a certain label looks like. This is done by maximizing the activation of a neuron in the output layer, rather than in the hidden layers as we did before. In Figure 3a we present the outcomes for the maximization of digit 1 for `DNN1`. No clear pattern is visible. In the process of analyzing these types of input examples we noticed that a high activation of one neuron does not necessarily implicate low activations of other neurons in the output layer, indicating a lack of confidence in certain classifications. Figure 3b shows the optimal input example for maximizing the

activation of neuron 1 minus the activation of all other neurons in the output layer. This input figure shows much more of a pattern, yet we still do not recognize digit 1. The two objective values used for these two images correspond with the *simple-* and *diff-objective* we proposed in section 3.3.1. The fact that we do see more of a pattern in the second image argues in favor of its corresponding objective function, which is something we experiment further on while constructing the heuristic for adversarial examples.



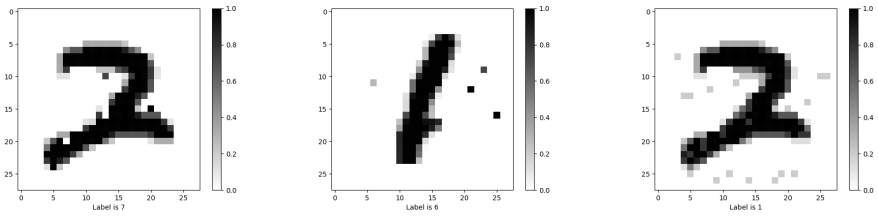

**(a)** Maximizes activation for digit 1

**(b)** Maximizes activation for digit 1 minus other activations

**Figure 3** Input examples that maximize activations in the output layer, for different objective functions. (`DNN1`)

### 5.1.2 Building adversarial examples

Adversarial examples are slightly perturbed inputs such that the DNN produces the wrong output classification. The following results relate to experiments for which correctly classified input figures are perturbed as little as possible such that it is receives a false a priori determined classification. Figure 4 shows adversarial examples that fool `DNN1`. In line with the results of (Fischetti & Jo, 2018), only a few well placed perturbations are necessary to cause the DNN to give the wrong classification. For 4a and 4b, no upper bound was set on the perturbation variables. For the examples in Figure 4c however, no pixel can be changed for more than the value of 0.2.




**(a)** fools `DNN1` to be a 7

**(b)** fools `DNN1` to be a 6

**(c)** fools `DNN1` to be a 7

**Figure 4** Input examples that fool `DNN1`

The following tables report statistics of the process of constructing adversarial examples for five different DNN architectures. Each statistic represents an average value over 100 runs for each DNN

and each model. For reliable comparisons of the computational performances of the models for different DNNs, we use the same set of 100 MNIST images for each run.

Table 2 shows the performance of the basic MILP formulation for this application and compares it with the model with tighter bounds. In this table, column "%solved" reports what percentage of the instances is solved to optimality. This strongly relates to the column "%gap", which shows the average optimality gaps when solving the problems. A gap equal to 0% corresponds with the solver having found an optimal solution. Columns "Nodes" and "Time(s)" give the average number of branching nodes and computing time in seconds, respectively. For `DNN4` and `DNN5`, some results are given within parenthesis. For these results, the improved model with weak bounds is used. We see from these results that when one wants to want to have a shorter preprocessing phase, these bounds serve as a good alternative.

When comparing the performance of the Basic model to that of the Improved model, it is obvious that the Improved model produces the best results. This conclusion was also made by (Fischetti & Jo, 2018). Apart from this similarity, we find many differences when comparing our table to that of the author's. The most striking difference is in the "%solved" column, namely that the number of instances that are solved within the time limit of 300 seconds is much lower. It should be noted however, that the overall low average optimality gaps indicate that our results were close to optimality. The second most difference seems to be the performance of the models applied to `DNN3`. More specifically, the performance for `DNN4` is better, whereas in the results of (Fischetti & Jo, 2018) this is not the case. It will require further investigation to identify the cause of this problem, which is beyond the scope of this paper.

**Table 2** Statistics of constructing adversarial examples, with a time limit of 300 seconds, averages for 100 different input examples.

| | Basic model | | | | Improved model | | | |
|---|---|---|---|---|---|---|---|---|
| | %solved | %gap | Nodes | Time(s) | %solved | %gap | Nodes | Time(s) |
| `DNN1` | 89 | 0.00 | 2,575 | 2.10 | 94 | 0.00 | 959 | 1.23 |
| `DNN2` | 66 | 0.61 | 85,055 | 53.37 | 80 | 0.00 | 14,092 | 13.78 |
| `DNN3` | 16 | 25.44 | 366,672 | 211.83 | 40 | 4.00 | 123,770 | 92.65 |
| `DNN4` | 18 | 19.67 | 324,213 | 200.02 | 44 | 0.00 | 80,561 | 65.83 |
| | | | | | (44) | (0.31) | (100,730) | (75.49) |
| `DNN5` | 1 | 73.67 | 342,071 | 293.36 | 5 | 34.07 | 189,230 | 250.94 |
| | | | | | (4) | (39.42) | (212,657) | (253.05) |

For the results in Table 3, we accept a solution to be sufficiently "solved" if it is guaranteed that it is within 1% of optimality. Especially for the improved model, we expect to find a large number of such solutions given the low gaps presented in Table 2. In contrast to the time limit of 3600 seconds that Fischetti and Jo (2018) imposed during these computations, we used one of 300 seconds to keep the time to calculate these statistics reasonable. Even after only 300 seconds, our results are close to those stated in the original paper. If one does not necessarily need an optimal example, but is content with an adversarial example somewhat close to this, working with these limiting settings seems to be the current best option.

**Table 3** Statistics of constructing adversarial examples, with a time limit of 300 seconds, averages for 100 different input examples. Computations are stopped when solutions are 1% or less away from the true optimum.

| | Basic model | | | | Improved model | | | |
|---|---|---|---|---|---|---|---|---|
| | %solved* | %gap | Nodes | Time(s) | %solved* | %gap | Nodes | Time(s) |
| `DNN1` | 100 | 0.54 | 2,575 | 2.00 | 100 | 0.47 | 961 | 1.23 |
| `DNN2` | 99 | 1.46 | 84,151 | 50.77 | 100 | 0.83 | 13,572 | 12.21 |
| `DNN3` | 47 | 24.82 | 363,578 | 207.88 | 85 | 6.20 | 124,814 | 92.04 |
| `DNN4` | 58 | 19.02 | 315,646 | 190.56 | 99 | 0.94 | 79,123 | 60.66 |
| `DNN5` | 4 | 76.60 | 297,723 | 292.33 | 34 | 34.86 | 189,362 | 246.41 |

*solved within 1% of optimality

## 5.2 A local search heuristic

In this subsection we present the performance of various configurations of the CDH. We perform multiple experiments for different choices of the building blocks presented and explained in 3.3.1. As our goal of building a heuristic is to be able to find adversarial examples for larger DNNs within reasonable computational time, we focus mainly on results for `DNN5`. However, we also perform experiments on `DNN1`, as we expect this to require less computational time.

To compare the performances of the heuristic methods to those of the best methods currently available, we summarize the relevant information of Tables 2 and 3 into Table 4. Moreover, we add the corresponding averages of the best objective values that the solver was able to find within the time limit. Based on the results in this table, we know that a good heuristic should find adversarial examples for `DNN5` in under 250 seconds, for an average total perturbation value of around 8.5. Important is the difference in non-solved models in this table and that of the heuristic. For the exact models, the required confidence ratio of 1.2 is included as a constraint in the model. Therefore,

it could still be possible to use a non-optimal solution as an adversarial example. For our heuristics, not finding a solution means that the required ratio is not reached. With this in mind, we decided not to put time limits on the construction of an adversarial example by the heuristic.

Table 5 shows results for the most basic setup of the CDH. In this Table, we see that for the fixed absolute perturbation solve rule, the results are the same for both objective functions. Based on the results, it seems that the heuristic with this solve rule finds examples faster and is less prone to get stuck in local optima than with the unbounded perturbation method.

**Table 4** Recap of relevant performances of currently known methods. Averages for 100 different input examples, with time limits of 300 seconds.

| | Improved model, gap = 0.0 | | | Improved model, gap $\leq$ 0.01 | | |
|---|---|---|---|---|---|---|
| | %solved | Time(s) | Obj | %solved* | Time(s) | Obj |
| `DNN1` | 94 | 1.23 | 8.71 | 100 | 1.23 | 8.71 |
| `DNN5` | 5 | 250.94 | 8.51 | 34 | 246.41 | 8.52 |

*solved within 1% of optimality

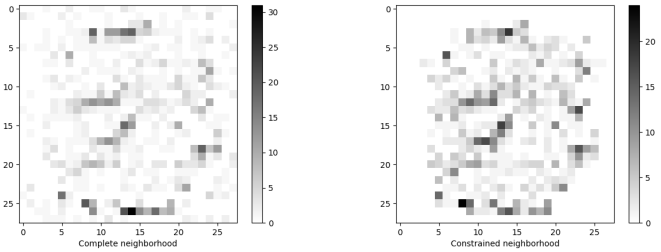
**Table 5** Comparisons of objective functions and perturbation methods, averages of solved instances. Searching in the complete neighbourhood, without local point escape methods. Applied to `DNN5`.

| | Unbounded perturbation | | | | Fixed absolute perturbation ($dj = 1$) | | | |
|---|---|---|---|---|---|---|---|---|
| Function | %solved | Time(s) | Obj | Iterations | %solved | Time(s) | Obj | Iterations |
| **simple** | 74 | 399.19 | 13.86 | 14.70 | 77 | 330.76 | 12.34 | 13 |
| **diff** | 75 | 334.31 | 10.81 | 11.88 | 77 | 330.36 | 12.34 | 13 |

It is obvious that these versions of the heuristic do not outperform the current methods available: both the average objective values as the computation times are higher than the desired targets for a good heuristic. The following subsections show and discuss results for the different choices in search and solve rules we explained in section 3.3.1. First, we investigate the effects of constraining the neighborhood by means of the probability mask. Then, we solve models for different parameter values of the fixed absolute perturbation solve rule to see if there exists a trade-off in objective value and computation time. Third, we try two different local point escape methods to increase the number of solved instances. Finally, we discuss the performance of the best found version of the heuristic.

### 5.2.1 Constrained neighborhood

In this section, we evaluate the effect of constraining the searching neighborhood by means of the probability mask. Figure 5 shows the number of times a neuron's activation was perturbed during the (successful) construction of 77 out of 100 adversarial examples. Figure 5a contains results for the search in the complete neighborhood, which can be compared to the results for the constrained neighborhood as presented in Figure 5b. From the results in these figures, we see that the overall pattern of perturbation remains the same, while the computational times are almost halved (from around 165 seconds to 90 seconds for the complete and constrained neighbourhood, respectively).




**(a)** Complete neighborhood search. Time(s) = 164.58, obj = 11.11

**(b)** Constrained neighborhood search. Time(s) = 89.91, obj = 11.39

**Figure 5** Dispersal of perturbations for different searching neighborhoods, sum of 77 instances. Experiments were performed on `DNN1`, with the diff-objective and with fixed absolute perturbations ($d_j = 1.0$).

### 5.2.2 Fixed absolute perturbation

In this section, we test different parameter values for the fixed absolute perturbation solve rule. Furthermore, we investigate whether the two types of objective functions behave differently for these experiments. In Table 5, we already observed that the use of this method with parameter value $d_j = 1$ results in lower computational times than when we use unbounded perturbation as a solve rule. Figure 6 shows that for lower values of $d_j$, the algorithm requires more time to find a solution while the objective value does not necessarily decrease. Again, the two objective functions produce similar results. For the highest values of the parameter, solution times are lower than the target. However, the objective values are at least 30 % higher and it should be noted that the non-solved instances are not included in these averages.

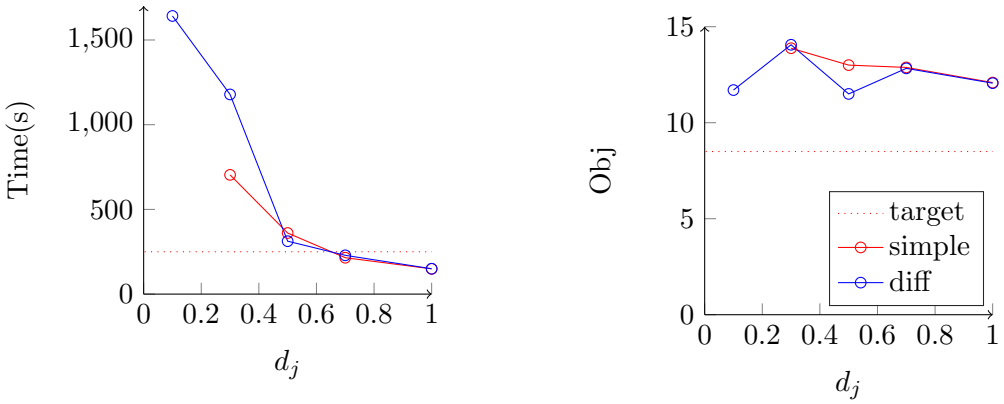



**Figure 6** Performance statistics for different values of the fixed absolute perturbation parameter. Experiments were performed on `DNN5`, while searching in the constrained neighbourhood with the diff-objective and the random local-point escape method. Averages computed for solved instances.

### 5.2.3 Local point escapes

In Table 5, we see that currently our best found configuration of the heuristic is able to find an adversarial example for 77% of the instances. In the other cases, the Local Search algorithm got stuck in a local optimum where it could not find any improvements according to the specified rules. In our methodology, we proposed two possible local point escape techniques: random perturbation and guided random perturbation. In comparison to the other experiments done in this paper, we ran the experiments for these two methods on multiple desktop computers with Intel Cores i5-8500U CPU at 1.60GHz and 16 GB RAM. Therefore, the results on computation time will only be used for comparisons with each other and not with other results in this paper.

Table 6 shows the results of various experiments for the two local point escape techniques. In this table, "$k$" represents the number of random perturbations performed in a single iteration of the escape method. We see that increasing $k$ often resulted in a higher solving percentage, but at the expense of a higher computation time and objective value. "#rand" stands for the total number of times the escape method is invoked for the instance set consisting out of 100 images. We see that a higher $k$ declines this number. Still, not a large difference in objective values is visible. A possible reason for this is that the random perturbations can also undo earlier added perturbations.

The guided random perturbation method fully activates one neuron and deactivates another. Table 6 shows results for two sets of parameter values for this method, where $p^h$ ($p^l$) is the probability mask-value a neuron should have to be fully activated (deactivated). Sadly, the results in this table do not give a clear image of the way these parameters influence the algorithm.

When comparing the results displayed in this table, we see that both escape methods are able to increase the number of solved instances. However, it is not clear which of the two methods works best in practice. In the next section, we present our preferred version of the heuristic for final comparisons. In this configuration we included the local point escape method which is shaded grey in the table, as this method solves a high percentage of instances with reasonable time and objective statistics.

**Table 6** Performance of two local point escape methods, averages of solved instances. Searching in the constrained neighbourhood with fixed absolute perturbation ($d_j = 1$), applied to `DNN1`.

| maxRand | %solved | Time(s) | Obj | #rand | %solved | Time(s) | Obj | #rand |
|---|---|---|---|---|---|---|---|---|
| | No escape method | | | | | | | |
| | 71 | 50.22 | 10.05 | | | | | |
| | Random perturbation method | | | | | | | |
| | k = 1 | | | | k=2 | | | |
| 10 | 81 | 60.44 | 11.49 | 233 | 79 | 53.03 | 10.78 | 236 |
| 15 | 81 | 55.19 | 11.02 | 330 | 84 | 57.98 | 11.83 | 298 |
| 25 | 83 | 60.05 | 11.65 | 524 | 86 | 88.99 | 11.63 | 489 |
| 75 | 88 | 77.88 | 12.17 | 1337 | 92 | 141.26 | 15.11 | 858 |
| | Guided random perturbation method | | | | | | | |
| | $p^h = 0.5$, $p^l = 0.01$ | | | | $p^h = 0.65$, $p^l = 0.001$ | | | |
| 10 | 77 | 83.01 | 10.76 | 252 | 77 | 52.12 | 10.97 | 251 |
| 15 | 82 | 96.52 | 10.99 | 368 | 84 | 57.75 | 11.62 | 317 |
| 25 | 87 | 61.89 | 11.86 | 528 | 90 | 63.81 | 11.85 | 464 |
| 75 | 95 | 74.60 | 13.34 | 757 | 89 | 57.46 | 11.56 | 993 |

## 5.3 Comparison and discussion

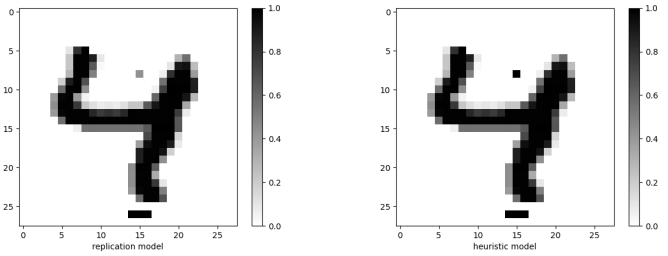
In this section, we combine all our main findings to construct a final version of the CDH. We analyze the results of this heuristic, and compare it to those of the methods used for Tables 4 and 5. For this final version of the heuristic, we search in the constrained neighbourhood, use the fixed absolute perturbation rule with $d_j = 1$ and escape local optima by means of the guided random perturbation method. The results for when we apply this heuristic to 100 instances for `DNN5` are presented in Figure 7. We see substantial improvements compared to the performance of the basic setup of the CDH as presented in Table 5. Moreover, for the instances that were solved, examples were found on within our time target of 250 seconds.

In practise, randomization methods were often invoked when the starting confidence ratio's for the false label were close to zero. In these cases, the local search method was not able to find

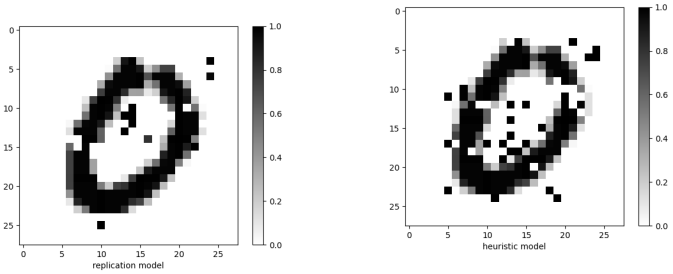
sufficient improvements as the desired solution was not in the close neighborhood. To illustrate the importance the starting conditions of the input figures for the CDH, we turn to Figure 7. In this Figure, the output of the CDH is compared to the output of the MILP model with tightened bounds and a 300 second time limit for two instances with different starting ratio's. From these results, we clearly see that the CDH is capable of approaching optimal solutions in a very short time for examples with favorable starting conditions, whereas it performs badly for initial solutions far distanced from the optimum.

**Table 7** Final configuration of the CDH applied to `DNN5`

| %solved | Time(s) | Obj |
|---|---|---|
| 84 | 185.45 | 10.96 |




**(a)** MILP: Time(s) = 132.2, Obj = 3.4**(b)** CDH: Time(s) = 45.4, Obj = 4.0




**(c)** MILP: Time(s) = 300, Obj = 14.7 **(d)** CHD: Time(s) = 688.3, Obj = 35.8

**Figure 7** Comparing the output of the CDH and the MILP model for different starting ratio's. The first adversarial example had a starting ratio of 0.665, the second 0.046.

## 6 Conclusion and future work

As stated in the introduction, the research in this paper was conducted to reach two goals. First, we aimed to replicate and validate the results of Fischetti and Jo (2018), who use a 0-1 Mixed Integer Linear Programming formulation to model Deep Neural Networks with ReLUs. While replicating

their results, we have seen that this formulation is of great use to gain insights in a given network's features and accuracy by means of creating optimized input examples. In particular, to investigate a network's accuracy and robustness the authors applied their models to adversarial machine learning. This is a particular area of machine learning that focuses on fooling a deep neural network with a slightly perturbed, initially correctly classified image. Based on our results for this matter, we come to the same conclusions as Fischetti and Jo (2018). For small networks, optimal adversarial examples can be constructed using the model in just a matter of seconds. However, for larger networks, computation times may excessive.

The second goal of this paper was to construct a heuristic capable of building adversarial examples for DNNs within reasonable computational time. We have outlined a framework of a local search matheuristic and presented multiple detailed interpretations. Although its performance was not optimal, we still believe we have established a great foundation for further developments in this field. We have shown how available training-data can be used to narrow down the searching area and therefore speed up the heuristic. Furthermore, our findings demonstrate that the use of random perturbation methods increases the probability of the heuristic successfully constructing an adversarial example.

To improve the performance of the heuristic proposed in this paper, further research should focus on increasing this probability of finding an adversarial example. As addressed in the previous section, the heuristic often failed to find a solution if the starting conditions were non ideal. We expect that the construction of a method for better initial solutions can increase the probability of finding an adversarial example for these cases, as well as decrease solving times for input examples with a better starting point.

During our work, we were limited by the fact that computing the reported statistics was computationally costly. Therefore, we were not be able to run experiments for a sufficient number of model parameters. Further research on this topic is therefore recommended.

## References


Belotti, P., Bonami, P., Fischetti, M., Lodi, A., Monaci, M., Nogales-Gómez, A., & Salvagnin, D. (2016). On handling indicator constraints in mixed integer programming. *Computational Optimization and Applications*, *65*(3), 545–566.

Cheng, C.-H., Nührenberg, G., & Ruess, H. (2017). Maximum resilience of artificial neural networks. In *International symposium on automated technology for verification and analysis* (pp. 251–268).

Chollet, F., et al. (2015). *Keras*. https://github.com/fchollet/keras. GitHub.

Dutta, S., Jha, S., Sankaranarayanan, S., & Tiwari, A. (2018). Output range analysis for deep feedforward neural networks. In *Transactions on computational science xxxviii* (p. 121–138). Transactions on Computational Science XXXVIII. doi: 10.1007/978-3-319-77935-5_9

Erhan, D., Bengio, Y., Courville, A., & Vincent, P. (2009). Visualizing higher-layer features of a deep network. *University of Montreal*, *1341*(3), 1.

Fischetti, M., & Fischetti, M. (2018). Matheuristics. In *Handbook of heuristics* (Vol. 1-2, pp. 121–153). Springer. doi: 10.1007/978-3-319-07124-4_14

Fischetti, M., & Jo, J. (2018). Deep neural networks and mixed integer linear optimization. *Constraints*, *23*(3), 296–309.

Fischetti, M., & Lodi, A. (2010). Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*.

Gigerenzer, G. (2008). Why heuristics work. *Perspectives on psychological science*, *3*(1), 20–29.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1) (No. 2). MIT press Cambridge.

Hagiwara, K., & Fukumizu, K. (2008). Relation between weight size and degree of over-fitting in neural network regression. *Neural networks*, *21*(1), 48–58.

IBM-Corporation. (2020). Ibm ilog cplex optimization studio 20.1.0 [Computer software manual].

LeCun, Y., & Cortes, C. (2010). MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. Retrieved 2016-01-14 14:24:11, from http://yann.lecun.com/exdb/mnist/

Lomuscio, A., & Maganti, L. (2017). *An approach to reachability analysis for feed-forward relu neural networks.*

Lourenço, H. R., Martin, O. C., & Stützle, T. (2019). Iterated local search: Framework and

applications. In *Handbook of metaheuristics* (pp. 129–168). Springer.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml*.

Park, J., & Boyd, S. (2017). General heuristics for nonconvex quadratically constrained quadratic programming. *arXiv preprint arXiv:1703.07870*.

Serra, T., Kumar, A., & Ramalingam, S. (2020). Lossless compression of deep neural networks. In *International conference on integration of constraint programming, artificial intelligence, and operations research* (pp. 417–430).

Serra, T., Tjandraatmadja, C., & Ramalingam, S. (2018). *Bounding and counting linear regions of deep neural networks.*

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). *Intriguing properties of neural networks.*

Tjeng, V., Xiao, K., & Tedrake, R. (2019). *Evaluating robustness of neural networks with mixed integer programming.*

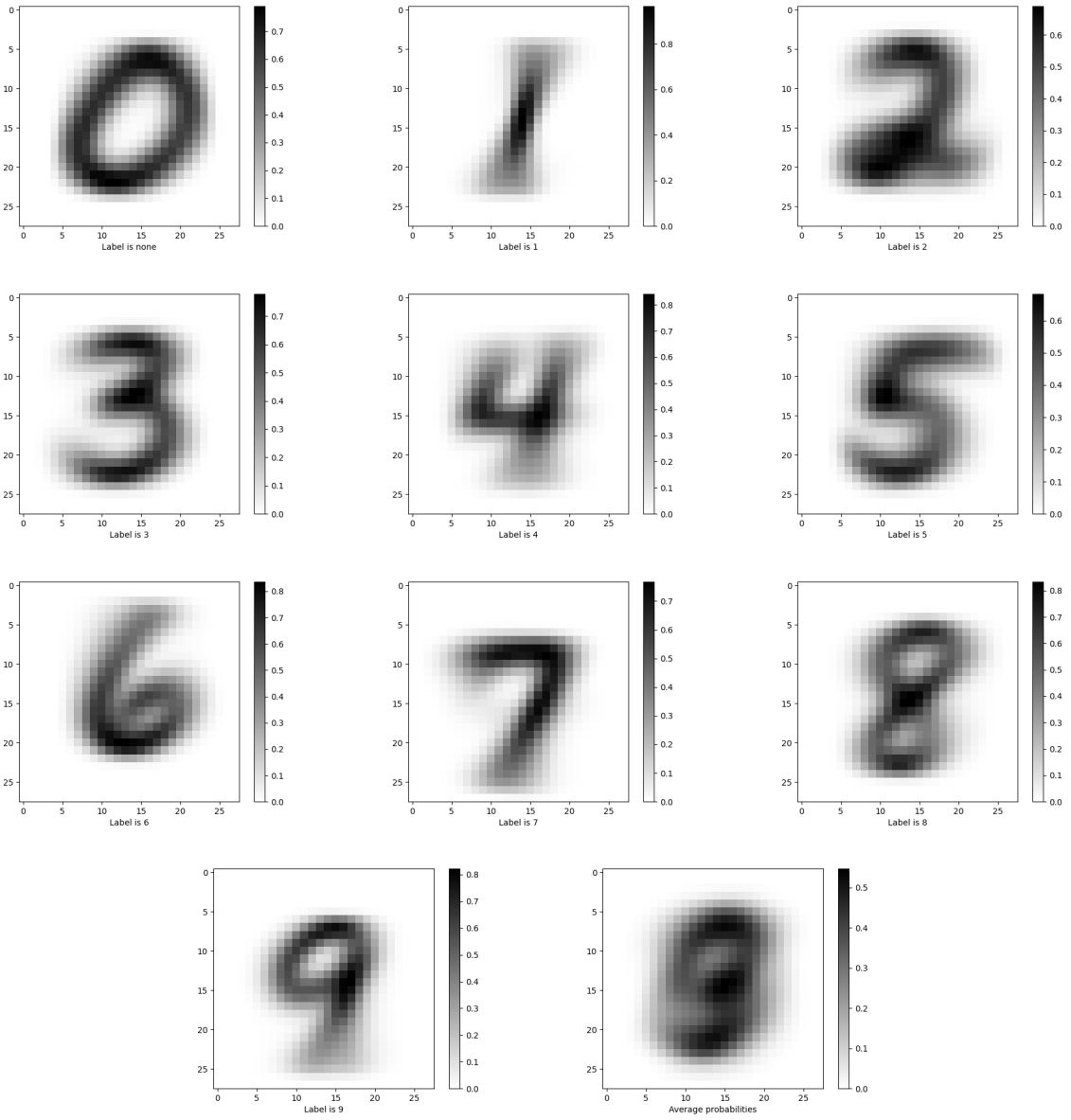
Van Rossum, G., & Drake Jr, F. L. (1995). *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.

Wilbaut, C., & Hanafi, S. (2009). New convergent heuristics for 0–1 mixed integer programming. *European journal of operational research*, *195*(1), 62–74.

Xue, M., Yuan, C., He, C., Wang, J., & Liu, W. (2021). Naturalae: Natural and robust physical adversarial examples for object detectors. *Journal of Information Security and Applications*, *57*, 102694. doi: 10.1016/j.jisa.2020.102694

# A Database analysis

For every possible label in the training set, we computed the average activation value to represent the probability that a neuron is activated. The following images are a visualization of the corresponding probability masks:



# B Parameters of the heuristic

**Table 8** A complete list of the parameters used in the CDH, together with the values we used in our experiments.

| Category | Symbol | Used value | Description |
|---|---|---|---|
| **Stopping rule** | - | 1.2 | Required false-label confidence ratio. |
| **Search rule** Constrained neighborhood | - | 0.01 | Minimum activation probability for a neuron to be included in the probability mask. |
| **Solve rule** Fixed absolute perturbation | $d_j^{fixed}$ | exp. | |
| **Solve rule** Local point escape | maxRand | exp. | Maximum number of times a randomization method can be invoked for a single instance. |
| **Solve rule** Local point escape | $k$ | exp. | Number of random neurons that are perturbed in a single run of the randomization method. |
| **Solve rule** Random perturbation | - | 0.5 | Threshold value for activating/deactivating a randomly drawn neuron. |
| **Solve rule** Guided random perturbation | $p^h$ | exp. | Minimum activation for a neuron to be drawn for a random activation. |
| **Solve rule** Guided random perturbation | $p^l$ | exp. | Maximum activation for a neuron to be drawn for a random deactivation. |
| **Solve rule** Guided random perturbation | maxIter | 500 | Maximum number of draws when the guided random perturbation method is invoked, to avoid infinitely cycling if no suitable neuron can possibly be found. |

"exp." indicates that we perform experiments on multiple values

# C Overview of programs

**Table 9** An overview of the programs used to compute our results. Contains descriptions of all classes, and highlights the most important methods.

| type | | name | description |
|---|---|---|---|
| `class` | | `MILP` | Used for the replication of the paper. Uses a `CPLEX` object to model a DNN into the MILP framework using our provided mathematical formulation, together with methods to retrieve information about solved instances. |
| `class` | | `Main` | Used for mainly the replication of the paper. In addition to the methods that follow, it also contains methods to write and read text files. |
| | `method` | `classify` | Uses the parameters of a given DNN to classify an input image. |
| | `method` | `compute bounds` | Computes upper bounds for the activation variables of a MILP, layer by layer. |
| | `method` | `feature-visualization` | Uses a MILP to maximize a certain activation of a neuron and writes the activations of the input layer into a text file. |
| | `method` | `construct adversarial examples` | Uses a MILP to construct a set of 100 adversarial examples. |
| `class` | | `MILPsub` | Used for the matheuristic. Very similar to the `MILP` class but with small adaptations in the way the disturbance variables are programmed. |
| `class` | | `MainHeuristics` | Used for all experiments regarding the CDH. Makes uses of the MILPsub class for small subproblems. Also uses guidedRand and Classification instances and methods to read and write text files. |
| | `method` | `matHeuristic-Constrained` | CDH formulation with unbounded perturbation, the constrained neighbbourhood and random local point perturbation. Can easily be adapted for complete neighbourhood search or guided random perturbation. |

| | | |
|---|---|---|
| method | constant-Heuristic | CDH formulation with fixed absolute perturbation, the constrained neighbbourhood and guided random local point perturbation. Can easily be adapted for complete neighbourhood search or random perturbation. |
| method | highest-without | Method used to compute the ratio's necessary for the stopping rule of the heuristic. Computes the highest value of the output layer, excluding the desired false label |
| method | constrain-inputlayer | Returns a boolean for every neuron in the inputlayer. If it is true, it can be included in the search area. If not, it should be skipped. |
| method | change-random-activation | Corresponds with the random perturbation method to escape local optima. |
| method | activation-probsets | A method for the guided random perturbation rule. Creates two sets of input neurons according to their activation and the value of the $p^h$, $p^l$ parameters, from which random neurons can be drawn later on. |
| method | change-guided-random-activation | Corresponds with the guided random perturbation rule. |
| class | Classification | This class was created for easy access to output layer activations. Stores the current classification, the array of output activations and contains a method to compute the ratio needed for the stopping rule of the CDH. |
| class | guidedRand | Creates objects for the guided random perturbation methods. Enables useful storing of both drawn neurons and their effects on the current image. |