Bachelor Thesis: Econometrie & Operationele Research (FEB63007)

Erasmus Universiteit Rotterdam

July 4, 2021

A Genetic Algorithm providing initial solution for a 0-1 MILP model representing Deep Neural Networks with ReLU activation functions

Author: Xander Maaskant (455169)

Abstract

Deep Neural Networks provide a great Machine Learning tool in pattern recognition. They can be represented through a 0-1 Mixed-Integer Linear Programming model introduced by Fischetti and Jo (2018). This model can point out weaknesses in the DNNs through the creation of adversarial examples. The current research successfully replicated the results and found these to be in line with the original. Moreover, we attempted to accelerate the computation times of the MILP formulation through the use of a hybrid solution algorithm based on a Genetic Algorithm. This algorithm is able to consistently provide the optimisation program with a feasible starting solution and provided promising results.

Supervisor: Bart van Rossum

Second Assessor: dr. T.A.B. Dollevoet

Contents

1 Introduction

Pattern Recognition and Machine Learning (ML) blend together perfectly. One can search the web with any combination of these terms and discover a substantial amount of scientific research regarding these topics. Deep Neural Networks (DNNs) are reckoned as one of the most popular ML techniques. Besides being an immensely popular technique, it is also an extremely effective one and it is continuously being researched and improved upon. However, these DNNs occasionally have a downfall. When training these DNNs, they tend to become very accurate at predicting patterns. Consequently so, the created network might be a bit too well fitted to the training data which will therefore trouble the robustness of the network. The result is that a slightly altered input or using an input with some obscurer values (adversarial examples), can fool the network and provide a wrong output. Mixed-Integer Linear Programming (MILP) can help out here with investigating these properties.

After training a DNN, its trained values can be used to construct a MILP formulation. Both methods consist of linear equations following up on each other, i.e. going from one layer to the next using a linear expression. The MILP formulation can help highlight the weak spots of certain DNNs. This is also what Fischetti and Jo (2018) showed, which is the basis of the current paper. They found an MILP formulation being able of providing the wrong output consistently with a slight alteration of the input. They do so using adversarial examples, these are able to fool the DNNs at hand. Moreover, they provided visualisation to further highlight the sore spots of the DNNs.

The current paper will therefore intend to replicate the results of Fischetti and Jo (2018) using the same data. The results of the original paper indicate that the computation times, of the MILP formulations, significantly increase with the expansion of a DNN, i.e. adding more nodes or layers in the hidden layers. They also state that these computation times make it difficult to investigate the larger DNNs and their properties. They, therefore, suggest that one can attempt to speed up these computation times by providing the optimisation program with an initial starting solution, i.e. a warm start, through a hybrid solution scheme. When provided with such a decent starting solution, it can decrease the computation times drastically. The current paper, therefore, provides an extension to improve upon the computation times of the bigger DNNs through the use of a Genetic Algorithm (GA). The GA will provide the optimisation program with such a starting solution.

Previous research provided different approaches for the suggestion of a warm start for an optimisation program, see Section 2. However, a GA is not explicitly used in a hybrid solution scheme with the intention to only provide an initial solution. This paper, therefore, contributes by using this semi-new approach (as GAs have been used before for finding MILP solutions) to investigate the possibility of accelerating the computation times.

The remainder of the paper is structured as follows. First, Section 2 will discuss the related works. It will elaborate on the important findings and conclusions of prior research on the topics of the current paper. Furthermore, a more brief motivation for the provided extension will be discussed. Section 3 will briefly elaborate on the data used for the research. After which the methodology will be discussed in Section 4 and 5. Elaborating on, the applicational use of the model and the use of the GA. Following up, the results will be presented and discussed in Section 6. Finally, Section 7 and 8 will provide the conclusion of the most important findings and the discussion about any obstacles encountered together with useful insights for future research, respectively.

2 Related work

The present paper mainly focuses on the replication of the paper from Fischetti and Jo (2018). They found a ML technique in the form of an 0-1 MILP formulation representing all the linear computations in a DNN based on ReLU activation functions. Hence, they could use the trained values of DNNs in the MILP formulation. Using the MILP formulation they were able to point out the hidden weaknesses of the DNNs through the creation of adversarial examples. Furthermore, they also proposed the use of bound tightening to accelerate the computation times of the models. However, they encountered that their MILP formulation still needs a lot of computation time when it represents a rather large DNN.

Around the same time, Serra et al. (2018) proposed a similar MILP model to represent a DNN, also based on ReLU activation functions. Likewise, they used bound tightening in their research. However, they were more interested in the dimensions and size of the DNNs and their implications on the performance accuracy of the specified DNN. In addition, Tjeng and Tedrake (2017) proposed a likewise model. Their research, again, made use of bound tightening. Matching the research of Fischetti and Jo (2018), as both were used to speed up the computation times of the models as well as reducing the gap between the best found and optimal solution, with success. Moreover, Tjeng and Tedrake (2017) were also able to provide a presolve algorithm which made the solving times even shorter. It is, therefore, a decent known method to represent a DNN through a MILP formulation and use it to investigate them.

As mentioned above, Fischetti and Jo (2018) and also Tjeng and Tedrake (2017) observed that larger DNN representations had sizeable computation times. The current paper therefore also researches the possibility of using an initial starting solution for the optimisation program, as suggested by Fischetti and Jo (2018). The optimisation program will start searching from this initial solution to the optimal solution which can speed up computation times. The use of such a so-called "warm start" is not a new concept as it was used in Rothberg (2007) through *polishing* and in Fischetti and Monaci (2014) through *proximity search*. Both these methods are based on neighbourhood searches. The current paper, however, will attempt to provide such an initial solution through a GA. GAs were initially designed to handle discrete values or bit strings representing real numbers, where it would handle these data vectors much like DNA and crossover by cutting and pasting from a certain point in the "code" or vector to form an offspring (Whitley, 1994). However, as also stated in Mitchell (1998), GAs can also be used on vectors only containing real numbers. This has become a frequent use. It is therefore possible to use the image data (see Section 3) and define this as the genetic code to be used in the GA.

In the past GAs have been used to find complete optimal solutions for non-linear MIP formulations by Yokota et al. (1996). Also, real-coded GAs have been proposed by Deep et al. (2009) and used to solve linear MIP formulations. One research that also proposed this method is done by Andrade et al. (2019). Their extensive GA was able to provide these optimal solutions, better than most other heuristics. The use of GAs in producing a warm start, however, is not that common. Seeing the evidence of GAs being able to provide optimal solutions, it should be able to also produce an initial starting solution. Since it is not in our interest to replace the MILP formulation, we instead want to speed up its computations through the use of a GA. The GAs that provided optimal solutions are quite extensive and are therefore not intended to produce quick starting solutions. Therefore, this paper will consider more simple GAs and focus more on their ability to quickly explore a neighbourhood of solutions (or so-called hyperplane sampling (Whitley, 1994)) to provide a feasible starting point.

3 Data

The current paper uses the same data as Fischetti and Jo (2018), to ensure accurately reproducing their results. The MNIST data set is obtained from LeCun and Cortes (2010). It contains handwritten numbers from 0 - 9 which are transformed to take on a pixelated appearance for useful implementation into ML methods. These now pixelated images can be transformed into data, with now each object representing a 2-dimensional (2D) image in a 28x28 grid. Each pixel in the grid is represented by a value, from 0 - 255. A higher pixel value implies a higher activation of the pixel in the image. Each 2D object is paired up with a label, representing the true classification of the handwritten numbers.

The data is naturally split up into a test and training set containing 10000 and 60000 observations respectively. All pixel values will be divided by 255 in order to only generate values between 0 and 1. This is a necessary step in order for the methods, described in the next sections, to properly function.

4 0-1 MILP formulation of a DNN

The main methods and formulations in this section are based on the paper of Fischetti and Jo (2018) as the present paper intends to replicate their results. Hence, first, the initial DNNs will be discussed (Section 4.1), after which the construction of the MILP formulation is explained (Section 4.2) and finally the applications of the MILP formulation will be presented. Some extra information is occasionally provided for certain methods to further explain the choices and assumptions made for the implementation of the models.

4.1 Deep Neural Networks

As mentioned in Section 2, the original paper starts by constructing the DNNs. From there, the trained values will be used in the construction of the MILP formulation that will follow up on this. Each DNN consists of $K + 1$ layers, these layers can be labelled as 0 to K. Layers 0 and K represent the input and output layer respectively. Each layer $k \in \{0, 1, \dots, K\}$ contains n nodes, labelled from 1 to n_k, indicating that the layers can vary in size.

As in the original paper, we can write UNIT(j, k) as the j^{th} node of the k^{th} layer. We then introduce $x_j^k \in \mathbb{R}$ for $j = 1, \dots, n_k$ to be the output of UNIT(j, k). The vectors x^0 and x^K correspond to the input and output vector of the DNN, respectively. The activation function, i.e. the function which defines the output of a certain layer k given the input of the previous layer $k-1$, is the *rectified linear unit* (ReLU). Hence, the output of each layer, for $k \geq 1$, can be computed through the following linear expression:

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

The *rectified linear unit* is defined for any real vector p as $ReLU(p) = max\{0, p\}$, where the maximum is computed component wise. The DNN can then be trained using Stochastic Gradient

Descent (SGD). Each DNN will be trained using 50 epochs.

Each DNN will be trained through back-propagation, as popularised by Rumelhart et al. (1986), using SGD (Bishop et al., 1995). For SGD, the derivative of the activation function is necessary. The theoretical derivative of the ReLU is defined as

$$f'(p) = \begin{cases} 1 & p > 0 \\ 0 & p < 0 \end{cases} \tag{2}$$

as was done in Straat (2018). That research also pointed out the issue encountered when $p = 0$ holds. In reality, however, we do not expect this to occur frequently. Moreover, as further suggested in Straat (2018), one can choose to select a value of either 0 or 1 when this occurs purely for implementational purposes. Liu et al. (2019) opted for $f'(p)$ to equal 0 when $p = 0$, the same will be assumed here during the training of our DNNs.

All DNNs will be trained in Python (Van Rossum and Drake, 2009) using the TensorFlow package (Abadi et al., 2015). This package is built for easy implementation of DNNs in Python and is perfectly compatible with the MNIST data set. It requires the input of the activation function per defined layer, the training method, and a loss function. The training method used is the back-propagation algorithm through the means of SGD using a learning rate of 0.01. The loss function used is the so-called Sparse Categorical Cross entropy function. This function computes the cross-entropy between the original labels and the outcome of the DNN to estimate the error. The DNNs are trained per epoch, each consisting of 10000 observations. This is done to prevent too much overfitting of the network.

All DNNs consist of an input layer and an output layer. The "body" of the DNNs that will be evaluated in this paper are as follows:

- DNN1: 3 hidden layers structured as: 8+8+8 nodes per hidden layer
- DNN2: 6 hidden layers structured as: 8+8+8+8+8+8 nodes per hidden layer
- DNN3: 4 hidden layers structured as: 20+10+8+8 nodes per hidden layer
- DNN4: 5 hidden layers structured as: 20+10+8+8+8 nodes per hidden layer
- DNN5: 5 hidden layers structured as: 20+20+10+10+10 nodes per hidden layer

4.2 0-1 Mixed Integer Linear Program

We want to get a valid 0-1 MILP representation that connects the input and output in the same way a DNN does. This entails that we have to represent the used activation functions from the DNNs, as a linear function to use in the MILP formulation. To get a valid 0-1 MILP model from a given DNN, i.e. using the trained weights and biases, the basic scalar equation

$$x = ReLU(w^T y + b) \tag{3}$$

is needed. This function can be rewritten as:

$$w^T y + b = x - s, \quad x \geq 0, \quad s \geq 0 \tag{4}$$

The equation in (4) is the linear representation of the ReLU function. As this function is defined as the maximum of 0 and its input, the x and s are needed to guarantee the linear expression will equal either one of these options. It is therefore needed to separate the positive and negative parts of the input into ReLU. However, the above equation does not provide a unique solution. Let $\mu \geq 0$, then one can obtain a feasible solution by adding μ to x and s ($x + \mu$, $s + \mu$). Therefore, $\mu = 0$ is a critical issue when rewriting the ReLU operator into a MILP formulation. In order to tackle this issue, Fischetti and Jo (2018) imposed the following solution. They introduced a binary activation variable z together with the following indicator constraints:

$$\left.\begin{aligned} &z = 1 \rightarrow x \leq 0 \\ &z = 0 \rightarrow s \leq 0 \\ &z \in \{0, 1\} \end{aligned}\right\} \tag{5}$$

These constraints can easily be implemented and solved using modern solvers such as CPLEX by ILOG IBM (2020).

With activation variable z_j^k for each NODE(j,k), the following 0-1 MILP formulation can be constructed from the earlier formulated DNNs, according to Fischetti and Jo (2018):

$$\text{minimise} \quad \sum_{k=0}^{K} \sum_{j=0}^{n_k} c_j^k x_j^k + \sum_{k=1}^{K} \sum_{j=1}^{n_k} \gamma_j^k z_j^k \tag{6}$$

subject to

$$\sum_{i-1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - s_j^k \qquad k = 1, \ldots, K, j = 1, \ldots, n_k \tag{7}$$

$$\left.\begin{aligned} &z_j^k = 1 \rightarrow x_j^k \leq 0 \\ &z_j^k = 0 \rightarrow s_j^k \leq 0 \end{aligned}\right\} \qquad k = 1, \ldots, K, j = 1, \ldots, n_k \tag{8}$$

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

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

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

$$x_j^k \geq 0 \qquad k = 1, \ldots, K, j = 1, \ldots, n_k \tag{12}$$

$$s_j^k \geq 0 \qquad k = 1, \ldots, K, j = 1, \ldots, n_k \tag{13}$$

$$z_j^k \in \{0, 1\} \qquad k = 1, \ldots, K, j = 1, \ldots, n_k \tag{14}$$

Since the DNNs are trained before the implementation of the above model, the weights (w_{ij}^k) and biases (b_j^k) are given and thus constant parameters. The same holds for the costs c_j^k and γ_j^k from the objective function. These can be defined according to the specification of the problem, i.e. for the problem at hand. The upper and lower bound can be set to 1 and 0, respectively, for the input layer ($ub_j^0 = 1$ and $lb_j^0 = 0$). Since the pixel values were standardised, the values will always be between these bounds. The constraints (7-8) and (12-14) define the ReLU output for each node. The remaining constraints (9-11) impose a lower and upper bound on the variables x and s. For $k = 0$, the lower and upper bounds apply to the original DNN input values of x_j^0. For $k \geq 1$, the lower and upper bounds can be defined as $lb_j^k = \bar{l}b_j^k = 0$ and $ub_j^k = \bar{u}b_j^k \in \mathbb{R}^+$.

The connection between the ReLU function and the translation into its constraints (7-8) and (12-14) can be explained through the following observation. When the left-hand side is smaller than zero, the corresponding x variable on the right-hand side is forced to zero through $z = 1$. Concerning the DNN outline, this corresponds to deactivating a certain node. Similarly, s is forced to zero through $z = 0$ when the left-hand side is greater than zero.

4.2.1 Bound tightening procedure

The model stated in Section 4.2, represents the simplest form of the 0-1 MILP formulation as the upper bounds for all the continuous variables (except the input layer) are continuous. Fischetti and Jo (2018) proposed a method that tightens these bounds. As mentioned before, Belotti et al. (2016) have shown that the tightening of the bounds in models such as the 0-1 MILP model (6-14) a tight upper bound is vital for the model to be practically solvable. Meaning, without these bounds the model is still solvable but has a very high computational time. With the use of the bounds, the computational time is decreased drastically.

The method used to find these upper bounds is as follows. The nodes in each DNN are scanned per layer ($k = 1, \ldots, K$) increasingly. For a current UNIT(j,k), all constraints and variables related to all other nodes in the current and/or subsequent layers are removed from the model. After this, the remaining model is solved twice, once for the maximisation of x_j^k and once for the maximisation of s_j^k. The found optimal values can then in turn be used to define a tight upper bound for both of the variables. The found values for the bounds in one iteration, i.e. for a specific layer, can be used to identify the bounds in the subsequent layers. This is due to the specific structure of the DNNs. After this process, one can save these bounds and use them for all further optimisation problems regarding the same DNN.

A further important note is that these bounds should be found using no input for the variables of the input layer, i.e. all x_j^0 variables. They do however depend on their respective lower and upper bounds (lb_j^0 and ub_j^0). As before, these are specified at 0 and 1 respectively for this problem.

As per the research of Fischetti and Jo (2018), the current paper will investigate two types of bounds. First of all, the exact bounds will be computed for all DNNs. Secondly, weaker bounds will be computed by setting the maximum computation time for each bound. These are also mathematically correct, yet do reduce the tightness on some bounds.

4.3 Applications

After the creation of the model, the models can be used as a foundation for applicational use. Two applications will be discussed: feature visualisation and adversarial examples.

4.3.1 Feature visualisation

Feature visualisation aims to find visual patterns for an input example which maximises the activation of a UNIT(j, k), for $k > 0$. The method is based on the works of Erhan et al. (2009). However, they performed the method through a greedy ascent method. This paper, like the original paper, uses the model (6-14) to find these maximising activation inputs. It can be achieved by changing the objective function into a maximisation function of the specified UNIT(j, k). The model will then create an input example that maximises the selected node. The input example can then be analysed on any visible patterns.

4.3.2 Adversarial examples

Adversarial examples help to point out any, perhaps hidden, weaknesses in the DNNs. We try to adjust the input ever so slightly, such that it still closely resembles the original input. However, in this process, it tricks the DNN into producing the wrong output. In other words, the DNN produces the wrong label to this new input due to the slight modification compared to the original.

As per the current setting, we have an original input figure $\tilde{x}^0$ that is correctly classified by the selected DNN. The correct label is denoted as $\tilde{d}$. As mentioned above, we want to find a new input x^0 that is wrongly classified as d, which is different from $\tilde{d}$. To achieve this, we need to alter the pixel values of the original input.

As in the original paper, the wanted output label d is also specified. This is done by setting $d = (\tilde{d} + 5)$ modules 10. We now have to ensure that the model selects d as the label for the altered input. To ensure this, the activation value of this new label (i.e. x^K_{d+1}) has to be at least 20% higher than any other activation in the output layer. This can be achieved by adding constraint (15) to the already existing MILP formulation (6-14).

Furthermore, we want to minimise the distance between x^0 and $\tilde{x}^0$. Hence, we want to produce the wrong label d by changing the least amount of pixel values of the original input $Tildex^0$. To achieve this minimisation, we introduce the variable d_j, representing the difference between $\tilde{x}^0$ and x^0. Therefore the new objective function for the MILP formulation becomes: minimise $\sum_{j=1}^{n^0} d_j$. Moreover, these continuous variables d_j must satisfy the constraints as in (16),

$$x^K_{d+1} \geq 1.2 x^K_{j+1} \qquad j \in \{0, , 9\} \backslash \{d\} \tag{15}$$

$$\left.\begin{aligned} -d_j \leq x^0_j - \tilde{x}^0_j \leq d_j \\ d_j \geq 0 \end{aligned}\right\} \qquad j = 1, \ldots, n_0 \tag{16}$$

One can now also easily add similar constraints, such as a maximum amount of pixels that can be changed or a maximum difference d_j per pixel. Besides analysing the results for the different DNNs, we will also provide images of these adversarial examples. This will also include examples containing a maximum difference for d_j of 0.2.

5 Genetic Algorithm

As mentioned previously, the current paper will provide a GA to investigate its possibilities to provide a "warm start" for the 0-1 MILP formulation and hoping the accelerate the computation times for the adversarial examples. The GA will need to provide a feasible starting point for the optimisation program, from where it will start optimising. When provided with a decent starting point, this can drastically improve the computation time (Fischetti and Monaci, 2014).

As described by Mitchell (1998) and Snaselova and Zboril (2015), the general set-up of a GA requires five parts: initial population, fitness function, selection, crossover, and mutation. A GA works much like natural selection in biological evolution, in which each generation adapts better to the current state to survive. Each generation can thus be seen as an iteration in the algorithm. In each generation, a population, consisting of individuals, reproduces to form a new population for the next generation. An individual represents a possible starting solution and is awarded a fitness score, which represents the quality of the individual. An individual is made up of so-called genomes, typically representing bit values (Mitchell, 1998). The genomes in the current paper represent the pixel values of the used images.

An initial population is necessary to start the algorithm. From there, the best individuals are selected for reproduction. This is known as selection. For reproduction, a crossover method is performed which determines the new offspring. Finally, a newly produced offspring can undergo some random mutation, creating the diversity within the population. This process is repeated until an individual is satisfactory, i.e. it provides a feasible starting solution. The algorithm can also be stopped after the maximum amount of generations is reached or if the algorithm stagnates, i.e. for a specified amount of generations no better individual was created. The algorithm keeps track of the best individual. If the possible solution is feasible, it can be used as a starting point for the optimisation program. The subsections below describe all the individual parts of the algorithm in detail and define the specifics for the algorithm used in the current paper.

5.1 Initial population

The initial population is the population for generation zero. The algorithm aims to find a feasible starting point for the 0-1 MILP model in creating adversarial examples, we can make smart use of pattern recognition abilities of DNNs. Therefore, we propose to first compute the average pixel activation per label over the entire training set. A minimum average pixel activation value is used (say p) and all pixels with a lower average activation value are set to zero. Hence, the pixels remaining have an average pixel activation greater or equal to p, resulting in an average pixel value v_j^i ($j = 1, \dots, n_0$ and $i \in \{0, \dots, 9\}$) for each different label.

Per adversarial example, we have an original input, as per Section 4.3.2. Hence, the initial population consists of two individuals: the original input $\tilde{x}^0$ and the average pixel activation v^i for the desired wrong output d, i.e. $i = d$.

5.2 Selection

In each generation, the best individuals are selected for the reproduction of the new population for the next generation. For the first generation, the initial population is selected in full for reproduction. Crossover $\tilde{x}^0$ with v^d until the desired population size (μ) is reached. This should provide the GA with a good starting point since the DNN is now very likely to produce the desired label dt due to the individuals already consisting of the (partial) pixel values of this wrong output.

For further generations, the amount of selected individuals depends on μ. Reproduction requires two individuals, with each pair of individuals producing two offspring (further elaborated on later).

Hence, the amount of selected best individuals, n, needs to be chosen such that $n^2 - n \geq \mu$. During reproduction, the individuals go in order from best to worst. Therefore, reproduction stops when the desired population size μ is reached, ensuring that the best individuals were able to reproduce. Leaving out only the less favourable individuals in the reproduction process.

As mentioned before, the algorithm can stagnate. When this occurs and NO individual has provided a feasible starting solution together with the maximum amount of generations not being reached yet, the algorithm restarts itself by creating a new initial population for the next generation. Do note that the algorithm does not get rid of the best-found individual. The algorithm, therefore, revives itself by providing the next generation with a completely new population created from the initial population, allowing the algorithm to keep searching for a feasible starting solution.

5.3 Crossover

Most crossover functions are based on genomes representing bits. However, our genomes represent real numbers in the range of $[0, 1]$. Thus, we make use of arithmetic crossovers. This crossover method uses selected parents P_k^{gen} and P_l^{gen} from the current generation to produce offspring O_k^{gen+1} and O_l^{gen+1} for the next generation, based on the formulas as in Köksoy and Yalcinoz (2008) and Yalcinoz et al. (2001). Hence, a random value $R \in [0, 1]$ is selected and creating offspring as follows:

$$\begin{aligned} O_k^{gen+1} &= R \cdot P_k^{gen} + (1 - R) \cdot P_l^{gen} \\ O_l^{gen+1} &= (1 - R) \cdot P_k^{gen} + R \cdot P_l^{gen} \end{aligned} \tag{17}$$

5.4 Mutation

After the offspring is created, some might mutate. As described in Whitley (1994), there are many ways to implement mutation methods. Most of those are based on binary or bit-by-bit mutation. Köksoy and Yalcinoz (2008) propose a method for real-valued individuals, using a probability for which each genome within the individual can mutate, i.e. a genome getting a new randomly selected value. To give more experimental freedom, our GA will use a mutation method with a mutation probability and mutation rate.

One can specify the mutation probability (δ) and mutation rate (ρ), according to the problem at hand. The former describes the probability of an individual mutating. The latter describes the rate at which an individual can mutate, i.e. how many genomes can mutate per mutation and thus ranging from 0 to 784. Mutation, therefore, happens with probability δ, after which ρ amount of random genomes are selected and each assigned with a new random value. The remainder of the genomes in the individual remains the same as in Köksoy and Yalcinoz (2008). The mutation provides new genetic data within the population creating differentiation throughout the GA and allowing it to perform the neighbourhood search.

5.5 Fitness function

Finally, each individual will be assigned a fitness score according to a fitness function, determining the quality of the individual. We need a feasible starting solution for the 0-1 MILP formulation. Hence, a good individual should oblige by constraint (15). Therefore, an indicator variable $\zeta_j + 1$ can be created:

$$\left.\begin{array}{ll} \zeta_{j+1} = 1 & \text{if } x^K_{d+1} \geq 1.2 x^K_{j+1} \\ \zeta_{j+1} = 0 & \text{otherwise} \end{array}\right\} \text{ for } j \in \{0,,9\} \backslash \{d\} \tag{18}$$

Moreover, the goal of the 0-1 MILP model when creating the adversarial example is to minimise the distances d_j. Hence, the entire fitness function becomes:

$$\sum_{j \in \{0,,9\} \backslash \{d\}} \beta \zeta_{j+1} - \sum_{j=1}^{n_0} d_j \tag{19}$$

Where β can be specified based on the problem at hand. The absolute maximum difference equals $\sum_{j=1}^{n_0} d_j = 784$, in case of no pixels activation and the new image activates all, or vice versa. Hence, the value of $\sum_{j \in \{0,,9\} \backslash \{d\}} \beta \zeta_{j+1}$ should in general be greater than this value, by selecting a proper value for β, in order to avoid individuals with a negative fitness as is generally preferred. The minus sign between the two sums derives from the goal being to minimise the distance. Hence, a feasible individual with a lower total distance is preferred and should therefore achieve a higher fitness.

In general, this fitness function should perform adequately. However, a possible downside for the GA is that it might have trouble with differentiating between infeasible solutions or near feasible solutions and actual feasible solutions, due to β being constant per constraint. Hence, when an individual only satisfies eight out of nine constraints, the algorithm might tend to begin optimising for the distance instead of searching for an individual that is close to satisfying all constraints.

6 Results

This section will first discuss the results of the feature visualisation (Section 6.1), following with the adversarial example results (Section 6.2), as done in the original paper of Fischetti and Jo (2018), and complete the section with the results of the GA (Section 6.3). The results were all computed on an 8-core MacBook Pro 2020, with 16 GB RAM and an M1 processor. As mentioned before, the DNNs were trained using Python (Van Rossum and Drake, 2009), together with all the visualisation of the images. All the other computations and results were achieved using the Java programming language (Arnold et al., 2005), which was extended with the IBM ILOG CPLEX 20.1 (ILOG IBM, 2020) solver for the implementation of the 0-1 MILP models.

To implement the 0-1 MILP models, all the DNNs needed to be trained first. The specified input and output layers consisted of 784 and 10 nodes, respectively, due to every image consists of 28x28 pixels and there are 10 different labels in total. All the DNNs were trained for 50 epochs, with each epoch consisting of a batch with 10000 random observations out of the possible 60000 in the MNIST training set. We found that increasing the number of observations per epoch did not increase the accuracy of the network, hence the batch size of 10000. After training, all DNNs produced an accuracy of 93 - 96% on the provided test set. After this, the obtained biases and weights were used to build the 0-1 MILP models as in (6-14). It should be noted that all the visualisations are created based on the results of DNN1, this model was able to solve almost all problems in about a second. The visualisations are mostly meant as clarifications and not as deep analyses, therefore the results from DNN1 are satisfactory.

6.1 Feature visualisation

The feature visualisation, much as in the original paper, did not provide any visible patterns. This was to be expected and is in line with the results of Fischetti and Jo (2018). The input examples maximising the activation of certain nodes can be found in Appendix A in Figure 4.

6.2 Adversarial examples

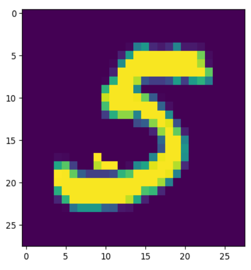
First, we will show and discuss the visualisation of some adversarial examples. Whereafter, the computational results will be discussed.

6.2.1 Visualisation

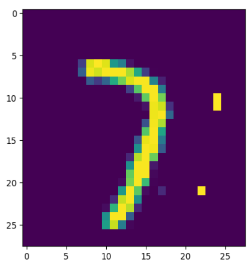
In Figures 1, 2 and 3 the adversarial examples are visualised. More examples can be found in Appendix B Figure 5, Appendix C Figure 6 and Appendix D Figure 7, respectively. A brighter (yellowish) pixel signifies a high activation and a more dull (dark purplish) pixel signifies little to no activation per pixel. Do notice that the images display the relative activation per pixel compared all the others, i.e. a bright pixel only indicates a high activation in comparison to the other pixels in the image.

Figure 1 displays two examples in which d_j can take on any arbitrary non-negative value. Hence, no limitation is imposed on the change of any pixel compared to the original input. It is though still limited to constraint (9). For Figure 2, a pixel was allowed to change with an absolute maximum of 0.2 ($d_j \leq 0.2$). The two figures lead to a remarkable difference, namely with a maximum imposed difference more pixels need to change to achieve the maximum activation of the specified wrong output node.

This difference is made more visibly obvious in Figure 3, where the left image displays the absolute distances with no limitations and the right image displays the absolute distances with a maximum change of 0.2. It can thus be observed that imposing a maximum change per pixel, entails changing more pixels. With the model having to change more pixels, a larger computation time is required. For a random sample of 100 observations from the test set, the MILP formulation for DNN1 created all the adversarial examples with no limitations and a maximum change of 0.2 on the distances in about 105 seconds and 225 seconds, respectively.

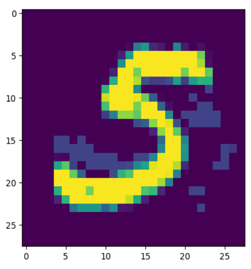


Label is 0

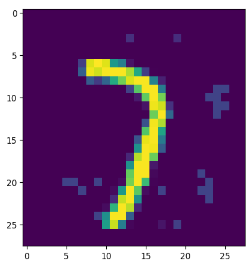


Label is 2

Figure 1: Adversarial examples created with the 0-1 MILP formulation with arbitrary distance, labels represent the output node with maximum activation



Label is 0



Label is 2

Figure 2: Adversarial examples created with the 0-1 MILP formulation with a maximum distance of 0.2 ($d_j \leq 0.2$), labels represent the output node with maximum activation

6.2.2 Computational results

The results in this section are based on 100 random runs per DNN. Each run is based on a different starting image from the MNIST test set. Per DNN these 100 data points can differ, due to not every DNN predicting the same data objects correctly and adversarial examples require a correctly identified data point as input. The 100 runs per DNN are, however, the same in each table for consistency purposes and therefore provide a better comparison between the different models.

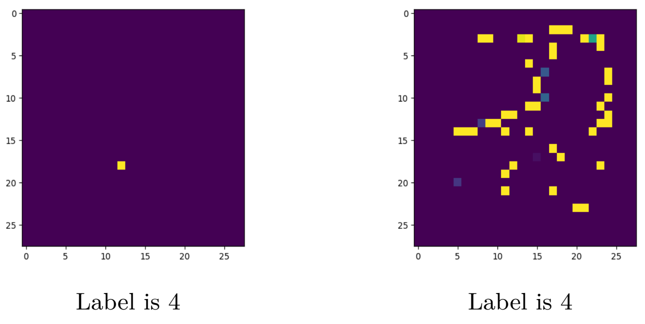


Figure 3: The absolute distances for the adversarial examples. The left and right image are the differences with arbitrary pixel change and a maximum pixel change of 0.2, respectively

This selection was achieved through seed setting when getting the random sample per DNN. Finally, each run had a time limit of 300 seconds.

Table 1 displays the results of creating the adversarial examples based on the basic model and the improved model (with exact bounds), as per Section 4.2.1. The %solved column reports the percentage of data points that have been solved optimally within the set time limit, i.e. the gap was less than 0.01% as the solver never returns a value of exactly 0%. The column %gap represents the average gap between the best found upper and lower bound per run. Moreover, the columns Nodes and Time (s) report the average amount of nodes and time in (wall-clock) seconds used by the solver per run, respectively.

The results show that bounds have a great positive effect on all of the statistics compared to the basic model. For DNN1 and DNN2 it had little impact, except for the computation time. For the others, it shows that the percentage of optimally found solutions (%solved) greatly increased. This is likely due to the strongly decreased amount of average nodes per run, also bringing down the computation time greatly. Both models are able to solve the problems optimally, however, the improved model greatly outperforms the basic model. These results fall completely in line with Fischetti and Jo (2018).

Table 1: Computational results of the basic and improved model

	Basic model				Improved model			
	%solved	%gap	Nodes	Time (s)	%solved	%gap	Nodes	Time (s)
DNN1	100	0.0	4710	1.0	100	0.0	752	0.4
DNN2	88	2.8	587482	93.0	100	0.0	38184	10.0
DNN3	82	4.2	749556	111.7	100	0.0	34458	9.1
DNN4	44	27.3	1230812	216.7	97	0.4	153331	39.5
DNN5	7	76.3	1030642	286.7	59	14.8	607972	187.8

With the great performance of the improved model, it also required quite the extra preparation time which is not included in the average time per run. The preparation time for the first three DNNs is negligible. However, for the two biggest DNNs (DNN4 and DNN5) this time is significant, as can be seen in Table 2. As mentioned by Fischetti and Jo (2018), this preparation time is only needed once per DNN and should therefore not be an issue in most applications. However, for some, it might become an issue when the network continuously adjusts itself and thus requiring the computation of the bounds each time.

Hence, as per Section 4.2.1, we will also look at weaker bounds with a time restriction on the computation time per bound. Table 2 presents the same format of results as Table 1, with an extra column (T. pre.) representing the total preparation time (in seconds) taken to compute the bounds per DNN. Obviously so, the preparation time for the weaker bounds is significantly less than for the exact bounds. Due to the weaker bounds being an upper bound on the exact bounds, the exact bounds outperform the weaker bounds to some degree. The same is encountered in Fischetti and Jo (2018). Noticeably so, both models significantly outperform the basic model. Appendix E shows more comparing results between the basic and improved model with weaker bounds, using the same time limit but with a minimum gap of 1% at which the found solution is said to be optimal.

Table 2: Performance comparison between the improved model with exact and weaker bounds

	Exact bounds					Weaker bounds				
	T. pre.	%solved	%gap	Nodes	Time (s)	T. pre.	%solved	%gap	Nodes	Time (s)
DNN4	400.5	97	0.4	153331	39.5	72.4	96	0.7	187416	47.4
DNN5	1955.2	59	14.8	607972	187.8	96.0	46	19.8	657434	203.1

6.3 Genetic algorithm

As per Section 5, first, the average pixel activation will be computed. This is done to help initialise the population of the GA. The minimum average pixel activation is set to 0.25, as this provided satisfactory results during experimentation. The pixels represented the shape of the assigned label properly. The pixel activation images for all labels can be found in Appendix F, Figure 8.

The GA was only implemented for DNN4 and DNN5, as these were the models with the greatest computation times. Before the GA can be implemented, the hyperparameters need to be set. These are the mutation probability and rate, the population size, and the maximum generation size. Experiments have been performed using DNN5 with a minimum generation size of 10 to ensure the GA runs for at least that amount of generations and to check for stagnation. The latter indicates the maximum amount of generations for which the GA is allowed to not improve, afterwards, the GA either stops when a satisfactory starting solution was found or a new initial population is created and it continues its search, see Section 5.2.

All hyperparameters have been experimented with to get a good impression of their effects. Firstly, the mutation probability and rate were experimented on. It turned out that an increase in mutation rate leads to an earlier stagnation of the algorithm. The mutation probability was found to be satisfactory at 0.6. The results of these experiments can be found in detail in Appendix G Table 6. Secondly, the generation and population size are experimented with. The highlighted results are observed in Table 3, the full results can be found in Appendix G Table 7. The highlighted results show that an increase in generation or population size does not provide more feasible starting solutions. It does show that an increase in generation size significantly decreases the average distance, in contrast to the population size which seems to have way less effect. As the MILP models will be solved on a 300 second time limit, the GA should not take up more than 10% of this. Hence, the presumed best combination of generation and population size is 1000 and 500 respectively. This combination generates an average distance of 14.43, the average optimal distance found by the MILP formulation is 5.77.

Table 3: Highlight results for different generation and population sizes

Generation size	Population size	# feas. solution	Ave. distance	Ave. generation reached	Time (s)
100	100	99	47.86	86	0.37
100	2000	99	37.98	100	8.68
1000	200	99	16.78	939	8.21
1000	500	99	14.43	983	21.44
1000	1000	99	13.45	993	43.23

Note: 100 runs were performed, using a mutation probability of 0.6 and a mutation rate of 1 pixel.

Hence, all the hyperparameters have been determined. The GA searched for feasible solutions for each run, thus a solution that satisfies constraint (15). Table 4 shows the results of the basic and improved model with weaker bounds when provided with a warm start by the GA. It can be seen that for almost all input images, the GA was able to provide such a starting solution. This is mainly due to the use of the average pixel activation. However, the averages are thus computed according to the feasible amount of initial starting solutions found out of 100 runs. Moreover, as before, all runs have a 300 second time limit. For each run, the time needed by the GA was subtracted from this time limit. Leaving the optimisation program with the difference as the time limit to find the optimal solution. This is done as the results are compared to Table 1 and 2. Due to the seed selection, the same sample per DNN was used in Table 4 as in Table 1 and 2.

First, the results in Table 4 show that the starting solution did not provide any computational advantages for DNN4. Probably entailing that this DNN is not complex enough and can be quite easily solved (as seen in Table 2) using bound tightening. On the contrary, the results do show a computational advantage for the model of DNN5. For the basic model, it provided more optimal solutions, a significantly lower relative gap, and a slight improvement in computation time. For the improved model, it improved significantly upon the amount of optimal solved problems and a slight improvement was made on the relative gap. The computation time was, however, not improved upon. These results imply that the computation time of the GA, especially for DNN4, might have been taken too long. This can be concluded as for DNN5 we are able to improve on all the other aspects, except for the computation time of the improved model. It might therefore be better to use a GA with a lower generation and population size to also improve upon the computation time consistently. Suggesting that it might be profitable to provide the optimisation program with a quicker starting solution instead of a better one when taking into account the computation time of this starting solution.

Table 4: Performance comparison between the models with and without a starting solution for the basic and improved model with weaker bounds

	Basic model				Imporved model (weaker bounds)			
	%solved	%gap	Nodes	Time (s)	%solved	%gap	Nodes	Time (s)
DNN4	37.1	29.0	1238675	235.5	95.9	0.7	200476	62.1
DNN5	10.1	70.7	1097920	281.2	52.5	18.5	623368	205.9

Note: For DNN4 and DNN5 there were 97 and 99 feasible starting solutions found with an average computation time of 17.5 (s) and 18.8 (s), respectively.

A decent starting solution can thus yield computational advantages. However, it can be seen that the current GA does not provide a good enough starting solution to really have an impact on the computation times. This is likely due to the relatively large computation time needed by the GA.

On the whole, our GA was able to consistently provide feasible starting solutions which proved to yield beneficial results. However, it could prove to be even more beneficial to further investigate the possibilities of accelerating the computational time needed by the GA to also consistently improve upon the computation time of the MILP model. Another possibility that might be interesting is to implement the current GA in even larger DNNs and check if it can accelerate those computations. Hence, the GA can yield more promising results even for larger DNNs instead of these relatively small DNNs.

One more noticeable observation when comparing the solutions provided by the GA and the optimal solutions found by the 0-1 MILP model, entails a sizeable difference in changed pixels. As shown in Section 6.2.1, the 0-1 MILP model often prefers changing only a few pixels. Whereas the GA solution seems to be changing quite a lot of pixels, mainly due to the random mutations in each generation. It can namely randomly select one pixel out of 784 each time it mutates. Hence, it is very likely that this will be a different pixel per mutation in each generation for the best individuals. Therefore, changing the value of a different pixel each time instead of slightly adjusting the value of the same pixel to explore nearby solutions like the optimisation program for the MILP formulation does. This effect of the GA was, nevertheless, attempted to be prevented through the chosen crossover procedure, as these bounded the values to the maximum of either one of the parents and therefore lowering the average deviation over the generations. This, however, ended up not negating the effect enough as can be seen according to the average distances in Table 3. Since the mutations are necessary for the neighbourhood search, for which the GA is intended, a point of improvement could be to look at the range of pixels the 0-1 MILP model is inclined to alter. Using this, one could restrict the GA in which pixels it is allowed to mutate and thus creating less deviation from the original image and thus providing a better starting solution.

7 Conclusion

The goal of the current paper was to replicate the results from the research of Fischetti and Jo (2018) and create a hybrid solution algorithm, based on a GA, to improve the computation times of the larger DNN representations of the 0-1 MILP formulations. The 0-1 MILP models were successfully implemented and the results between the current and original paper were very similar, differing only slightly in values but being able to draw the same conclusions. The main differences are the computation time for the exact bounds and the number of nodes needed by the optimisation program, both possibly due to implementation differences or the use of the newer version of IBM

ILOG CPLEX (ILOG IBM, 2020). The 0-1 MILP models therefore successfully enlightened the weaknesses of the DNNs by creating the adversarial examples and being able to trick the DNNs into producing the wrong output.

The implemented GA was able to provide feasible starting solutions for almost all input images. However, it did not significantly improve upon the computation times. For the largest DNN, it was able to improve upon the optimally solved problems as well as the relative gap, consistently. It, therefore, did show decent potential. Thus, when further improved and investigated, it might also be possible to speed up the computation times significantly and consistently for bigger DNN representations through the 0-1 MILP models.

Concluding, the results are in line with the original paper and the provided hybrid solution algorithm, based on a GA, proved to be promising.

8 Discussion

During the research for the current paper, some limitations and obstacles were encountered which were not mentioned in the original paper of Fischetti and Jo (2018). One obstacle was the relative lack of description and clarification by the authors. Therefore making it quite the challenge to replicate their results using the same ML techniques for the DNNs. As a result of this, the current paper attempted to clarify more on some of the choices and assumptions needed for the implementation of the DNNs and the 0-1 MILP models. For further research one could try different assumptions in an attempt to replicate the results of the original and current paper.

As mentioned by Fischetti and Jo (2018) and also experienced in the current paper, the computation times of the bigger DNNs limited the broadening and expansion of the MILP model. The GA slightly improved upon the computational results, although not being completely successful, it proved to be able to produce feasible starting solutions consistently. Therefore, it is an interesting method to explore for further research by investigating its possibilities for even larger DNNs as well as researching the mutation process to further optimise the GA and provide even better starting solutions for the MILP model. One final interesting topic for further research is the implementation of the fitness function and letting it be more distinguishable between infeasible and near feasible solutions.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Andrade, C. E., Silva, T., and Pessoa, L. S. (2019). Minimizing flowtime in a flowshop scheduling problem with a biased random-key genetic algorithm. *Expert Systems with Applications*, 128:67–80.

Arnold, K., Gosling, J., and Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

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

Bishop, C. M. et al. (1995). *Neural networks for pattern recognition*. Oxford university press.

Bouman, P. (2018). JavaCplexExample. https://github.com/pcbouman-eur/JavaCplexExample.

Deep, K., Singh, K. P., Kansal, M., and Mohan, C. (2009). A real coded genetic algorithm for solving integer and mixed integer optimization problems. *Applied Mathematics and Computation*, 212(2):505–518.

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

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

Fischetti, M. and Monaci, M. (2014). Proximity search for 0-1 mixed-integer convex programming. *Journal of Heuristics*, 20(6):709–731.

ILOG IBM (2020). V20.1: User's manual for CPLEX. *International Business Machines Corporation*.

Köksoy, O. and Yalcinoz, T. (2008). Robust design using pareto type optimization: a genetic algorithm with arithmetic crossover. *Computers & Industrial Engineering*, 55(1):208–218.

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

Liu, Y., Wang, X., Wang, L., and Liu, D. (2019). A modified leaky relu scheme (mlrs) for topology optimization with multiple materials. *Applied Mathematics and Computation*, 352:188–204.

Mitchell, M. (1998). *An introduction to genetic algorithms.* MIT press.

Rothberg, E. (2007). An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Serra, T., Tjandraatmadja, C., and Ramalingam, S. (2018). Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pages 4558–4566. PMLR.

Snaselova, P. and Zboril, F. (2015). Genetic algorithm using theory of chaos. *Procedia computer science*, 51:316–325.

Straat, M. (2018). *On-line learning in neural networks with ReLU activations.* PhD thesis.

Tjeng, V. and Tedrake, R. (2017). Verifying neural networks with mixed integer programming. *CoRR*, abs/1711.07356.

Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.

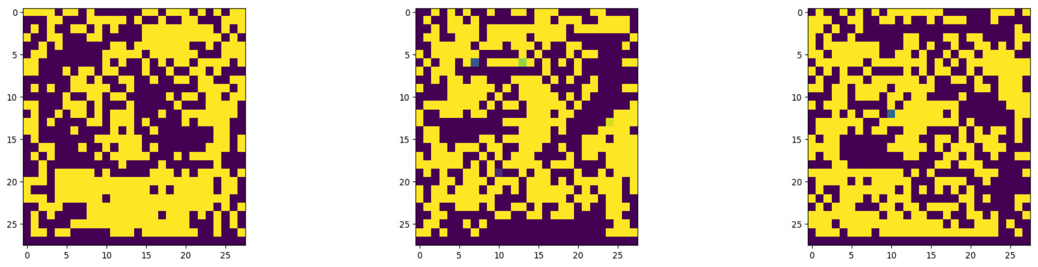
Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85.

Yalcinoz, T., Altun, H., and Uzam, M. (2001). Economic dispatch solution using a genetic algorithm based on arithmetic crossover. In *2001 IEEE Porto Power Tech Proceedings (Cat. No. 01EX502)*, volume 2, pages 4–pp. IEEE.

Yokota, T., Gen, M., and Li, Y.-X. (1996). Genetic algorithm for non-linear mixed integer programming problems and its applications. *Computers & industrial engineering*, 30(4):905–917.

Appendix A - Feature visualisation results

This appendix depicts the input examples of the performed feature visualisation of DNN1, these can be seen in Figure 4. The fifth node was maximised for layers one to three in the figure from left to right. For the maximisation of each node, a different label was found for the created input example. All three images show no visible pattern. Hence, no further conclusions can be drawn from these images.

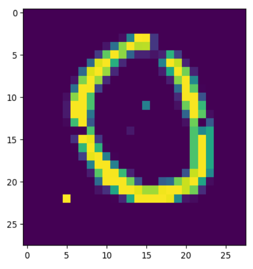


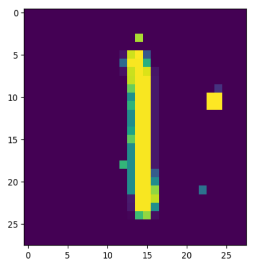
Label 0 Label 2 Label 3

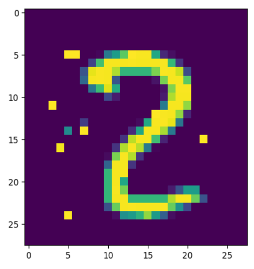
Figure 4: Input examples maximising the activation of node 5 from layers 1, 2 and 3 of DNN1, respectively

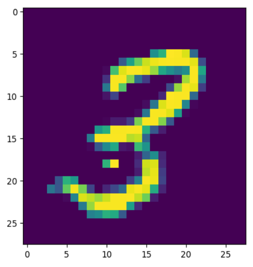
Appendix B - Adversarial examples with arbitrary distance

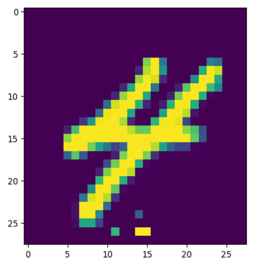
This appendix depicts an example of all the other labels as in Figure 1. Hence, the adversarial examples are created through the 0-1 MILP formulation for DNN1 in which there is no maximum imposed on the change of a pixel.

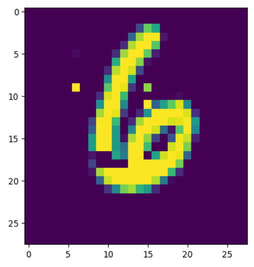

Label is 5

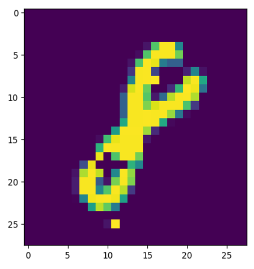

Label is 6


Label is 7


Label is 8


Label is 9


Label is 1


Label is 3

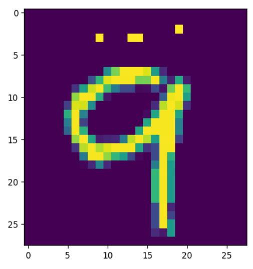

Label is 4

Figure 5: Adversarial examples created with the 0-1 MILP formulation with arbitrary distance, labels represent the output node with maximum activation

Appendix C - Adversarial examples with a maximum distance

This appendix depicts an example of all the other labels as in Figure 2. Hence, the adversarial examples are created through the 0-1 MILP formulation for DNN1 in which a pixel cannot change with more than 0.2, i.e. $d_j \leq 0.2$ for $j = 1, \ldots, n_0$.

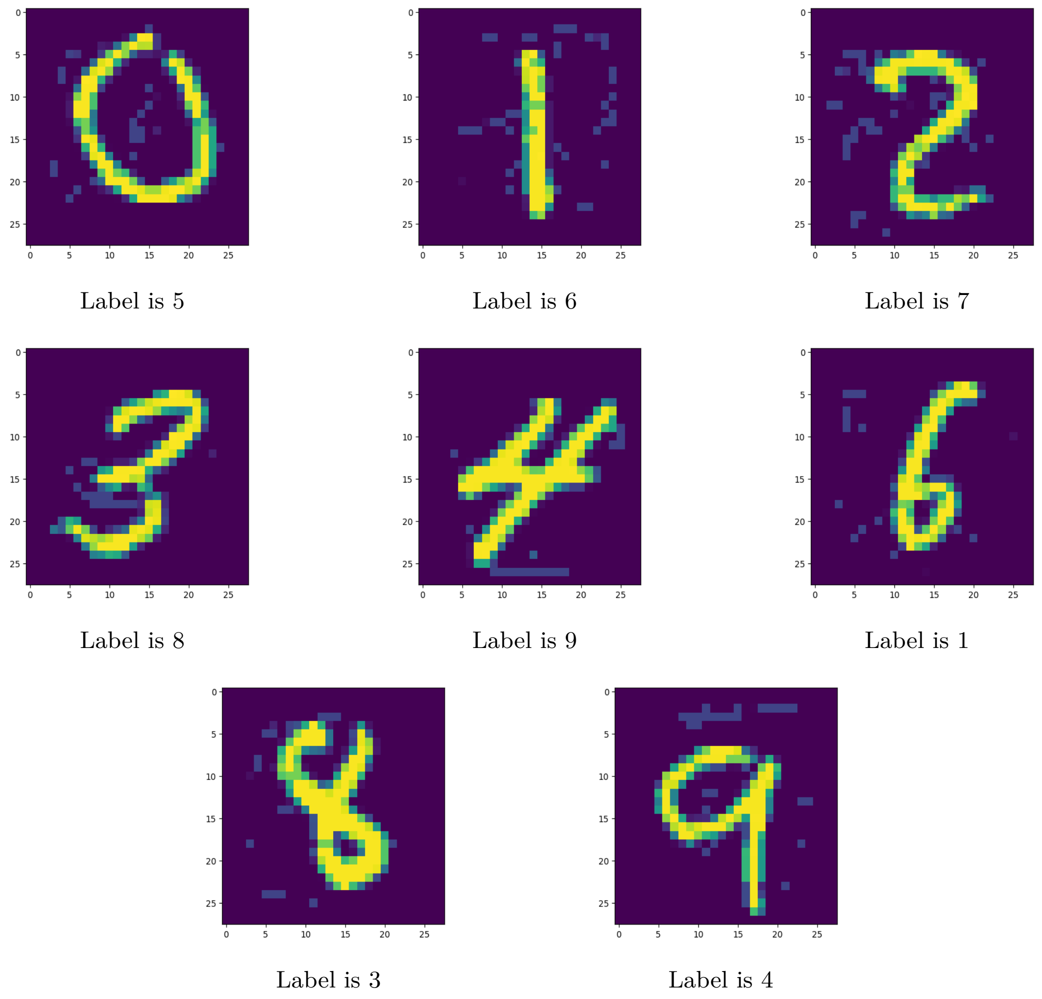
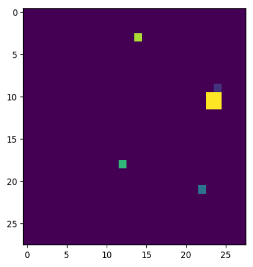


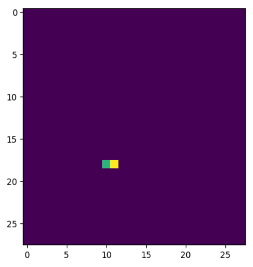
Figure 6: Adversarial examples created with the 0-1 MILP formulation with a maximum distance of 0.2 ($d_j \leq 0.2$), labels represent the output node with maximum activation

Appendix D - Adversarial examples absolute distances

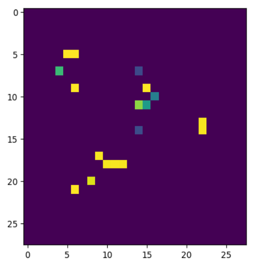
This appendix depicts more examples of the other labels as in Figure 3 for adversarial examples with arbitrary pixel change and maximum pixel change. The figures depict the actual change in pixels for the adversarial examples. As mentioned in Section 6.1, the pixel brightness is relative to the other pixels in the same image. The labels are again the output nodes with maximum activation.



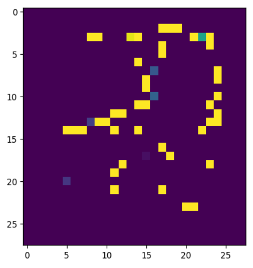
Label is 6



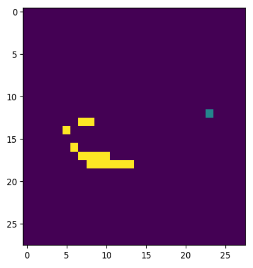
Label is 8



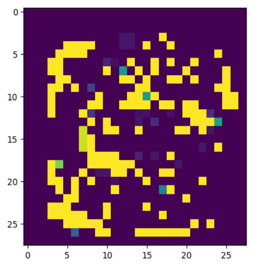
Label is 1



Label is 6



Label is 8



Label is 1

Figure 7: The absolute distances for the adversarial examples. The left and right image are the differences with arbitrary pixel change and a maximum pixel change of 0.2, respectively

Appendix E - Comparison of models with a minimum gap

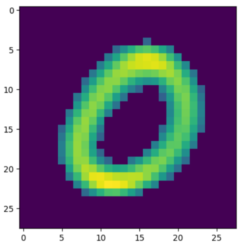
This appendix shows the results as done in Fischetti and Jo (2018). It compares the results of the basic model and improved model with weak bounds. Both models are gain evaluated with a time limit of 300 seconds. However, a minimum gap of 1% is imposed. Thus, a solution is found to be optimal if the gap between the best found upper and lower bound is less or equal to 1%. The results in Table 5 show that the improved model was able to almost optimally compute all the solutions for all DNNs but DNN5 within one minute. The column # represents the number of runs that were not able to be solved optimally and reached the time limit. The other columns are presented in the same format as Table 1 and 2. Do note that the average gap can be greater than 1% due to some of the runs reaching the time limit. It can also be lower than 1% due to the solver jumping over the 1% optimality when exploring a new solution. The results are in line with the original paper.

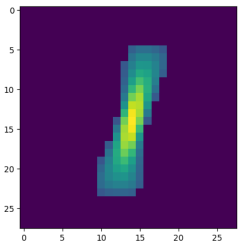
Table 5: Performance comparison of the basic and improved model with weaker bounds to get solutions with a 1% error or less

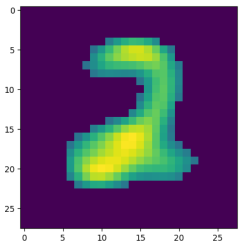
	Basic model				Improved model (weak bounds)			
	#timelim	Time (s)	Nodes	%gap	#timelim	Time (s)	Nodes	%gap
DNN1	0	1.1	4738	0.6	0	0.4	763	0.4
DNN2	9	91.2	565329	3.1	0	10.0	40557	0.9
DNN3	14	115.2	782186	4.0	0	10.4	40746	0.9
DNN4	60	225.7	1191784	30.6	4	43.1	156640	1.7
DNN5	96	290.3	1002894	80.3	48	201.7	671469	18.4

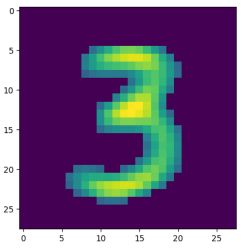
Appendix F - Average pixel activation

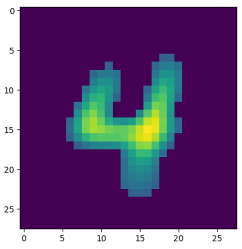
This appendix depicts an example of all the labels for the average pixel activation. These figures are created by taking the average activation per pixel across 60000 observations in the training set. After taking the average, only pixels with an average activation of 0.25 or greater ($\bar{x}_j^0 \geq 0.25$ for $j = 1, \ldots, n_0$) are kept, i.e. an average pixel activation of at least 25%. Noticeably so, the images represent the assigned label properly.

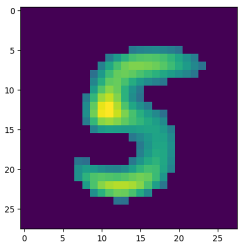

Label 0

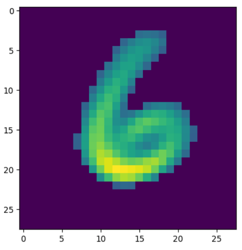

Label 1

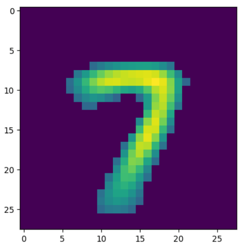

Label 2

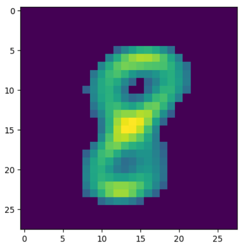

Label 3


Label 4


Label 5


Label 6


Label 7


Label 8

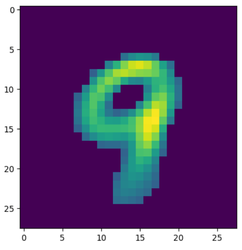

Label 9

Figure 8: Average pixel activation over the training set, with at least a 25% average pixel activation

Appendix G - Genetic Algorithm experimentation results

This appendix shows the in-depth results of the experimentation with the GA through the use of different hyperparameter settings. These results were generated for DNN5 for 100 runs. Table 6 shows the results using a (maximum) generation and population size of 100, keeping these constant allows for a good comparison. The goal was to investigate the effect of the mutation probability and rate (amount of pixel to be mutated). The results show that an increase in the mutation rate leads to an early convergence of the GA, indicating it is not able to find a good local optimum. Increasing the mutation probability indicates a lower distance. Hence, from this table, it can be concluded that a 0.6 mutation probability with a mutation rate of 1 pixel.

Table 6: Genetic algorithm results for different mutation probabilities and rates

Mutation prop.	Pixels mutated	Ave. distance	Ave. generation reached	Time (s)
0.1	1	54.48	81	0.36
0.1	2	57.26	37	0.16
0.1	5	59.03	13	0.06
0.1	10	59.17	11	0.05
0.2	1	52.64	74	0.32
0.2	2	54.41	64	0.28
0.2	5	58.42	17	0.07
0.2	10	58.88	12	0.05
0.4	1	48.68	90	0.40
0.4	2	52.61	68	0.30
0.4	5	58.16	16	0.07
0.4	10	59.15	12	0.05
0.6	1	47.86	86	0.38
0.6	2	50.73	83	0.37
0.6	5	57.23	27	0.12
0.6	10	58.95	12	0.05

Note: 100 runs were performed based on a generation and population size of 100.

Next, the generation and population size are investigated. These results are presented in Table 7. For these results, the mutation probability and rate were held constant at 0.6 and 1, respectively, as proven to generate the best possible results as per Table 6. It can be seen that a higher (maximum) generation size decreases the distance significantly. The effect of increasing the population size is also evident, yet it does not have as much effect. To determine the best combination of these hyperparameters for the warm start one needs to take into account that the maximum time for each run is 300 seconds. Therefore, the GA should not take up more than 10% of this as this would decrease the allowed time for the optimisation program too much. Hence, the thought-out decision is made to use a (maximum) generation size of 1000 with a population size of 500 individuals.

Table 7: Genetic algorithm results for different generation and population sizes

Generation size	Population size	# feas. solution	Ave. distance	Ave. generation reached	Time (s)
100	100	99	47.86	86	0.37
100	200	99	44.16	100	0.85
100	500	99	41.60	100	2.14
100	1000	99	39.83	100	4.32
100	2000	99	37.98	100	8.68
200	100	99	40.59	170	0.74
200	200	99	35.78	199	1.73
200	500	99	32.45	200	4.32
200	1000	99	30.32	200	8.69
200	2000	99	28.53	200	17.39
500	100	99	29.22	419	1.83
500	200	99	23.23	494	4.31
500	500	99	20.02	496	10.77
500	1000	99	18.27	500	21.78
500	2000	99	17.08	500	43.63
1000	100	99	23.54	725	3.17
1000	200	99	16.78	939	8.21
1000	500	99	14.43	983	21.44
1000	1000	99	13.45	993	43.23
1000	2000	99	12.89	1000	87.19

Note: 100 runs were performed, using a mutation probability of 0.6 and a mutation rate of 1 pixel.

Appendix H - Code

This appendix provides a general overview of the code used to obtain the results in the tables and figures presented in this paper. As mentioned in the paper, two types of programming languages were used: Python and Java. We will first discuss the code for Python as this code produces data that is needed in Java. Finally, the code for Java will be discussed. The following subsections provide a general summary, for more in-depth information please refer to the actual code files.

Python code

The code written in Python consists of six classes, a *main* class and one class per DNN. The *main* class contains functions to read the data and transform these into CSV files to use in Java. It also has a function to visualise given data objects through CSV files, these objects should contain 784 values (pixels) per row. Finally, this class contains a function which trains all the DNNs at once.

The classes per DNN (*DNN1*, *DNN2*, *DNN3*, *DNN4* and *DNN5*) contain an initialisation function, requiring the batch size, training data, etc. Moreover, it also has a function that creates the model per specified DNN. In this function, the model is trained, evaluated on the test set and its biases and weights are exported to CSV files to be used in Java.

Java code

For the Java code, the classes are divided into different packages as this presented an easy overview. Hence, the discussion will follow the style and thus discuss the code per package. The classes within the packages will be discussed in alphabetical order and the main intention behind each class and its main functions will be discussed.

Default package

This package contains two classes, *Main* and *Test*. The *Main* class is the mainframe for the MILP problems, in this class, one can run a variety of different functions. Each of these is designed for a different part of the results provided in the current paper. These functions are general and allow the user to specify some input. Most functions save the results to either a CSV or text file. The *Test* class is a class to run the results of feature visualisation and experiment with the genetic algorithm together with its MILP formulation. DNN5 is the model upon which the GAs experiment. The feature visualisation is defined on DNN1.

Methods package

This package contains the classes that define a neural network. In the code, the DNNs are defined using directed graphs as these have very similar structures. Furthermore, also all computational functions per DNN are defined in this package together with the functions to compute the bounds used in the models.

The classes *Node*, *DirectedGraphArc* and *DirectedGraph* make up the frameworks of the DNNs. The classes *DirectedGraphArc* and *DirectedGraph* were provided to us by Bouman (2018). The *Node* class has been created to represent a node in the graph, containing the following information per node: layer k, node number j, and the bias. If the node is for the input layer the bias is set to 0. The other two classes, create the directed graph. On the graph, one can find all the nodes, create arcs between nodes, get the in and out arcs from a specific node, etc. One can thus create the nodes and add them to the graph and based on the information in the created nodes one can create the arcs for the graph as well. These arcs contain the in and out node as well as information over the arc. In our case this will be the weights from one layer to the next per node, as the value of the next node can be defined as the sum over the activations times the weights of all nodes of the previous layer plus the biases of the current node.

The class *DataFunctions* contains function to compute all sorts of results per DNN. This class contains a function to read the biases and weights for a DNN, as well as the data objects and (adversarial) labels. Moreover, this class creates all the model objects for all runs of a DNN. It prints the necessary information to either a CSV or text file, to analyse the results or create the images using the previously mentioned code in Python.

The class *Bounds* is used to compute the bounds for a DNN, it can do so for the exact bounds as well as for weak bounds with a specified time limit per bound. It saves the bounds to a CSV file.

Networks package

This package contains the classes for each DNN. The functions within each DNN are generally the same, except for the naming of CSV files or destinations of these. Furthermore, the class *DNN1* contains a function to perform feature visualisation and the classes *DNN4* and *DNN5* contains functions to perform the hybrid solution algorithm with the GA.

Each class thus contains a function per method necessary for the computation of the results or in-between steps, such as the computation of the bounds per DNN. In those functions, it calls upon the needed data objects which are read or loaded in through the use of the *DataFunctions* class.

For the function that runs the results, it creates the needed graph, after which it starts running the results through the created models and saves the needed output statistics which are later saved to a text file. Some new created input images, distances, etc are saved to CSV files. They can later be used for visualisation purposes.

Models package

This package contains the code for all the MILP models used in the paper. It differentiates per needed use of the MILP model. Hence, each different use has a different class, i.e. for the basic and improved model two different classes are created, yet for the exact and weaker bounds incorporated in the model, no new class was created. These classes call upon the CPLEX solver, by first creating a CPLEX object and then one by one adding in the variables and constraints. The variables are saved to a map as these need to be accessed later on in the creation of the constraints as well as obtaining the solutions from the model.

Each class thus consists of functions to create the variables, constraints and also to set the input, acquire the desired solution, and any other needed information after the model is solved.

GeneticAlgorithm package

This package contains all the code for the genetic algorithm. It contains an object class *Algorithm* for the entire algorithm, which in turn calls upon the other classes *Crossover*, *Selection*, *Population*, *Individual* and *DNNComputation*. There is one other class in the package, *ActivationNodes*, this class computes the average pixel activation used in the initial population of the GA. Yet, this computation is done beforehand.

The *Algorithm* class, thus runs the algorithm and provides the best-found individual. *Individual* is the class representing an individual, it contains its pixels (genomes), the current generation, current individual number of the generation, fitness score, and if it is satisfactory or not. *Population* is in turn the object class for the population per generation consisting of individuals. We can sort the population and get back the best individual. It, therefore, has its own mergesort and compare functions to manipulate the individuals. The *Selection* and *Crossover* classes define the named after procedures. The *Crossover* class also contains a function for the mutation.

The class *DNNComputation* performs all computations regarding the outline of the DNN. It thus has functions to compute the fitness, output layer, and starting values of a best-found individual. To do so it defines some computational functions for matrix and vector manipulations as well as the ReLU function.