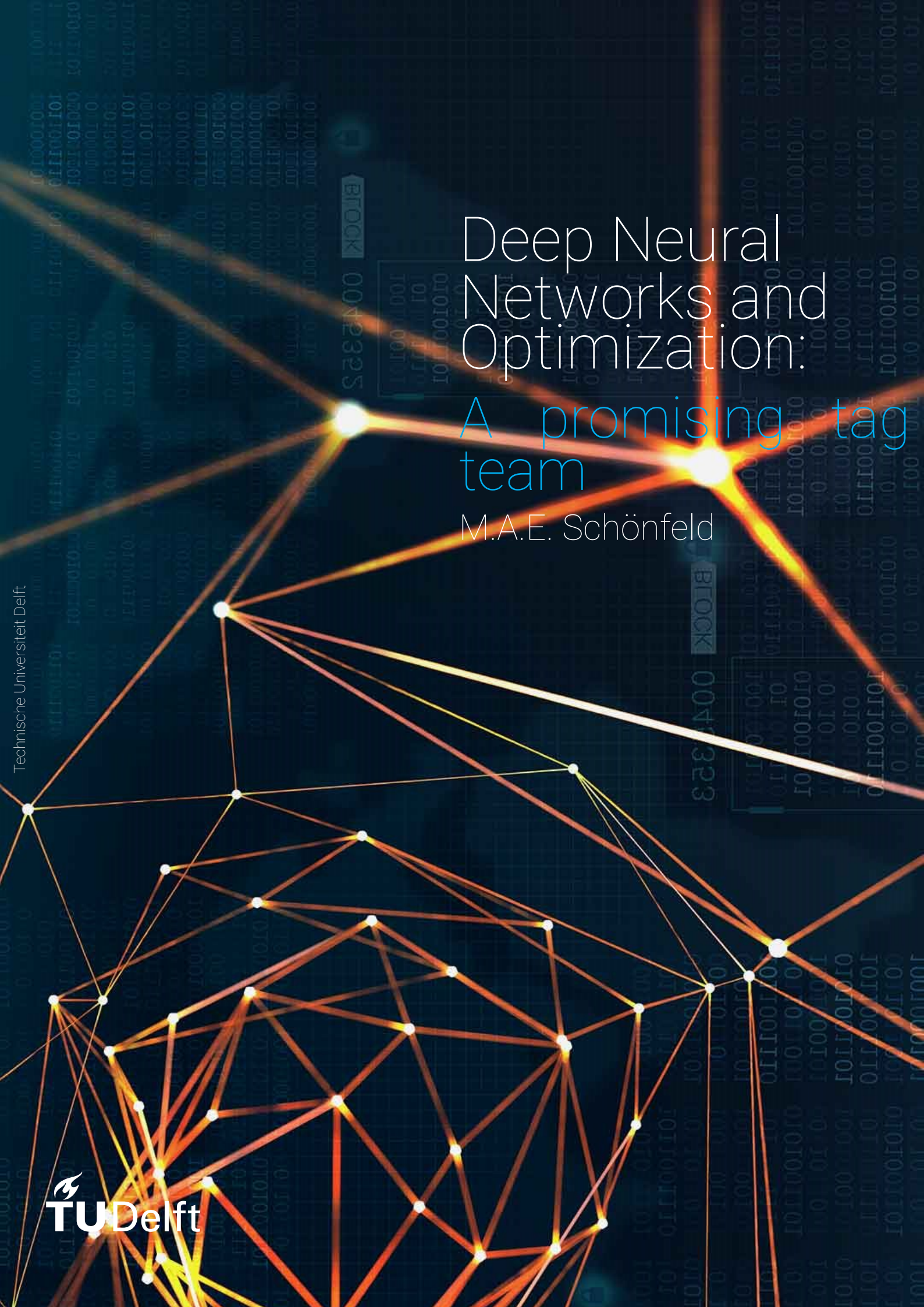

Deep Neural Networks and Optimization:

A promising tag team

M.A.E. Schönfeld

Technische Universiteit Delft

TU Delft

Deep Neural Networks and Optimization:

A promising tag team

by

M.A.E. Schönfeld

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Tuesday January 17, 2020 at 13:30.

Student number:	4474147	
Project duration:	September 1, 2019 – January 17, 2020	
Thesis committee:	Prof. dr. ir. K. Aardal,	TU Delft, supervisor
	Drs. E. M. van Elderen,	TU Delft
	Dr. N. Yorke-Smith,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This report is about an upcoming field of research: the combination of machine learning and optimization. It was written to fulfill the graduation requirements of the applied mathematics bachelor at the TU Delft. I was engaged in writing the report from September 2019 until January 2020.

Because both fields are not extensively taught during the mathematics bachelor I spent a lot of time on learning the basic principles of both fields. This means that I myself have learned a lot during this project but that there was not a lot of time left to do actual research. There will be some added details to an existing model, some restrictions not mentioned or explored before and some minor experiments. I do believe that this combination of fields has a lot of promise and hope that it will be researched further. I therefore hope that this report will not be seen as a valuable source to the applications of the field, but rather a comprehensive summary understandable for students of both backgrounds.

During the project I could trust my supervisor Prof. Dr. Ir. Karen Aardal to always be available for any type of questions or other guidance and I am very grateful for such an involved supervisor. I would also like to thank my friends, family and especially my partner for supporting me and always being available for friendly sparring to give me direction in this project.

I hope you find this report perspicuous and that you learn something along the way.

M.A.E. Schönfeld
Delft, January 2020

Contents

1 Introduction

Machine learning has been a computer sciences buzzword for years. The technology has a lot of potential and a huge number of applications that spoke to people with and without knowledge of computer sciences. Image, text and speech recognition, social profiling, computergames, everything seemed possible. Machine learning is not as much in the spotlight now since the rise of blockchain technology, but the field is still relevant and used widely. The field concerns itself with 'teaching' computers to make choices or recognize data by repeating experiments or looking at data, and then making changes to itself to achieve better results.
Now consider the field of optimization. It is not as well known outside of the mathematics community but incredibly important nonetheless. It has a miriad of uses: flight scheduling, ambulance placement and route planning are a minimal selection of all the problems that Optimization gets used in. Optimization is about solving problems as quickly and optimally as possible, looking at ways to speed up the process or finding the solution with the lowest cost.
Both fields have an extensive range of methods and applications but have not been combined a lot before. This report takes a versatile topic within machine learning, namely deep learning and combines it with a classic problem from Optimization. This model was first proposed in 2017, meaning that is a very young line of research. This report will further explore this crossover and look into the performance and possible applications.
There will first be an introduction to both fields to make the report understandable for students from either mathematics or computer sciences faculties. The model will then be established and explained in detail. There will then be time devoted to looking into applications of the model and seeing how they perform in experiments. Lastly the results of the experiments will be discussed and their implications.

2

Prerequisites

Deep Neural Networks and Mixed Integer Linear Programming are topics in Computer Sciences and Mathematics respectively that do not have a lot of overlap at first glance and have not often been researched in tandem. Whether this can be attributed to few people having extensive knowledge about both subjects or the lack of possible research routes is debatable. The field is upcoming however and it is not unlikely that if more people research the topic some big discoveries could be made. This report does explore the combination of these fields and to accommodate people from either faculties a short introduction is appropriate. This short explanation only attempts to give some brief intuitive understanding however and is not meant to give a theoretical background because this is not needed to understand the line of research. For further reading about Deep Neural Networks and Machine Learning the book *Understand Machine Learning: from theory to algorithms* by Shai Shalev-Shwartz and Shai Ben-David [14] is recommended and for further reading about Mixed Integer Linear Programming the book *Integer programming* by Laurence A. Wolsey [16] is recommended.

2.1. Deep Neural Networks

A Deep Neural Network (hereafter referred to as a DNN) is a topic studied within the field of Deep Learning, Deep Learning being a sub-topic of machine learning. Machine learning is the study of algorithms and statistics to make computers perform a task, by learning from data to see what would be the best course of action. Deep learning is then a more specific category of algorithms that use a layered structure. A DNN has layers of neurons, where all neurons are connected to all neurons in the next layer. Layers are only directly connected to the layer before and the layer after, with all connections going in only one direction: from the input layer towards the output layer.

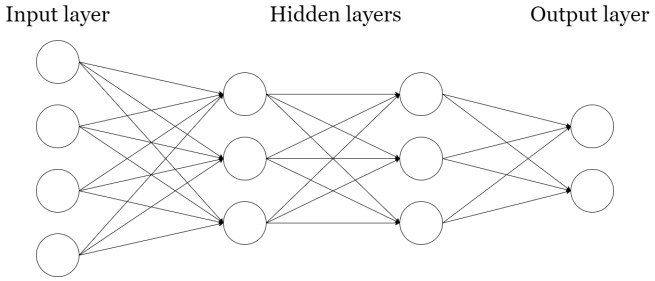

Figure 2.1: Basic visual representation of a neural network

The network takes an input such as an image or sound recording represented by its pixel values or frequency levels. The **input layer** is then fed through the layers of the DNN and ends with the **output layer**. The output often says something about the input when properly **trained**. DNNs are a form of machine learning and are trained in a similar way: a dataset with labels of the data is fed through the network and the network is rewarded for right answers and punished for wrong ones.
Consider an image of a black handwritten number over white background. We take this image from the

MNIST database [11] , which is a dataset of 70,000 handwritten images paired with their correct label. This dataset is often used for elementary neural networks because of its size and because it is open source.

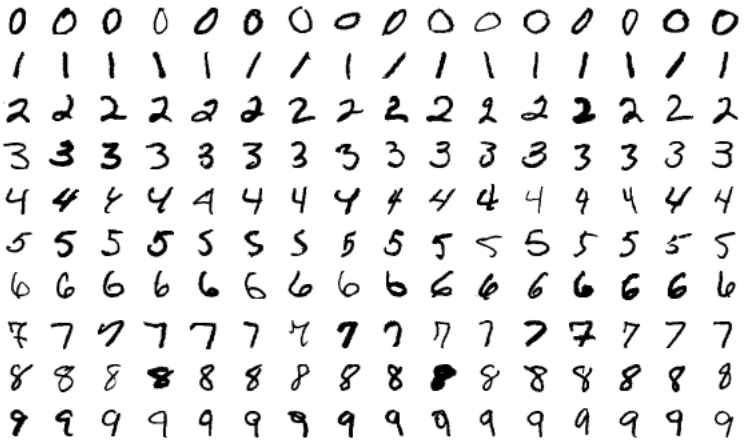

Figure 2.2: Some samples from the MNIST database.

The image is represented as a vector of pixel values between 0 and 1 where 1 means black, 0 means white and all values in between are grey. Most humans can easily read the character, but how would a computer see this number?

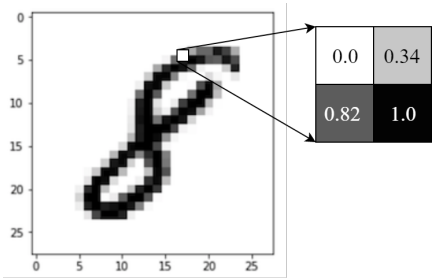

Figure 2.3: A detail from an MNIST image with greyscale pixel values.

A nine has a small circle with a curved stick on the bottom, so the program could expect black pixels in certain areas. In this way an eight and a 9 would have the same recognition in the top half of the picture, but differ in the bottom half.

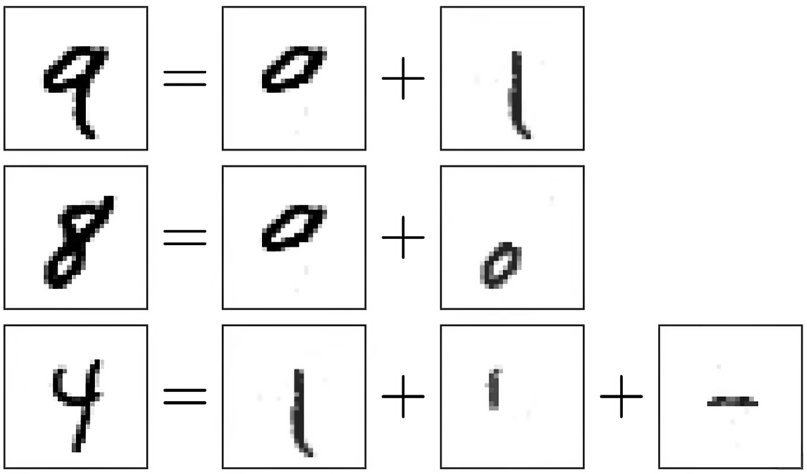

Figure 2.4: MNIST numbers split into components.

So consider a pixel in the top half of the picture that is black in the loop part, the computer should be rewarding it with a positive value for both 8 and 9, but give a negative reward for 1 because 1 does not have a loop at the top. This value is called a **weight**, because it weighs the pixel value against an expected value for a certain number. Weights are usually denoted as w_i or w_{ij} The minimal value before it gets a reward is then called a **bias**. Lastly this value is run through an **activation function**, usually denoted with σ, to pump up the positive rewards and negate the negative rewards. In the end all these rewards are tallied up and the number that has received the most rewards in this process is recognized to be the number pictured.

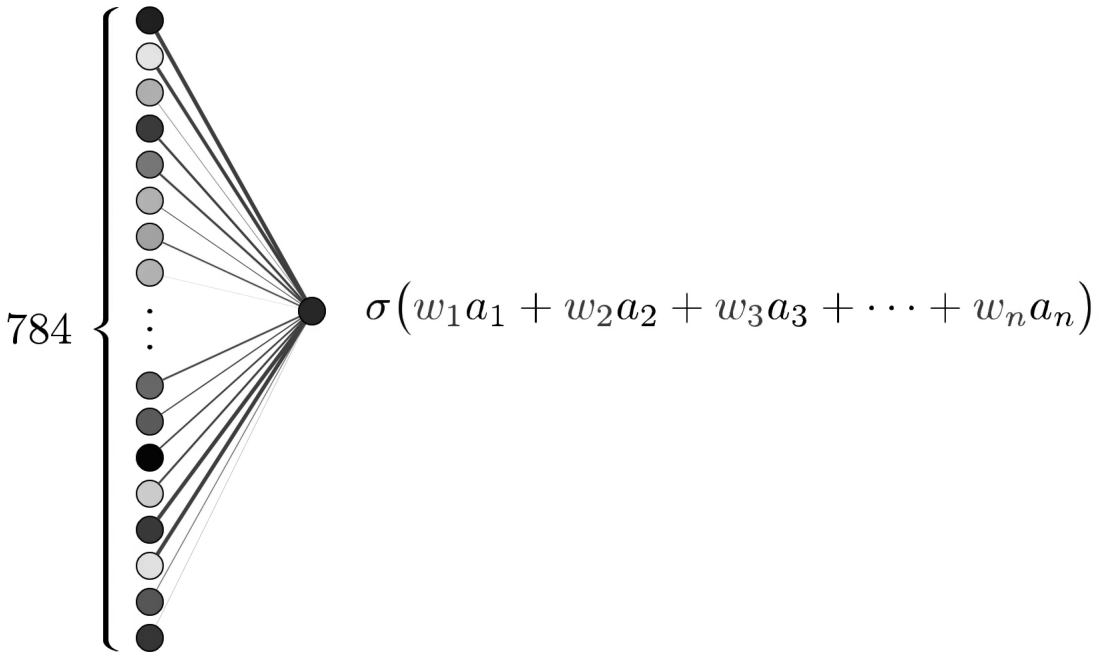

Figure 2.5: A visual representation of the input layer being connected to one single neuron in the next layer. Note how this single neuron receives information from all neurons in the previous layer.

If a picture has 28x28 pixels, all these weights and biases need to be determined. In the case of just an input and output layer $784x10 = 7840$ weights and 10 biases would need to be determined. It is of course not needed to state that doing so by hand would be an unproductive method and a waste of time. So the desire is to have a system that takes in all of the 784 pixels and evaluates them with the weights and biases to recognize a number, but the system needs to be able to find the appropriate weights and biases by itself. Even better would be if the first vector could be run through the process to recognize loops and sticks and then recognize the number by seeing what kind of combination of loops and sticks it is. A system of multiple **layers** would therefore be an obvious choice.

2.1.1. Mathematical formulation

With this intuitive explanation the mathematical explanation for a DNN can now be established. Consider a network of K layers. Every layer k except the last one has a matrix of weights W^k and a bias vector b^k. The starting vector x^0 is then fed through the first layer to an output vector of the first layer with

$$x^1 = \sigma(W^0 x^0 + b^0) \tag{2.1}$$

with σ an activation function such as one of the well known activation functions: the sigmoid function or the rectified linear unit (hereafter referred to as $ReLU$).

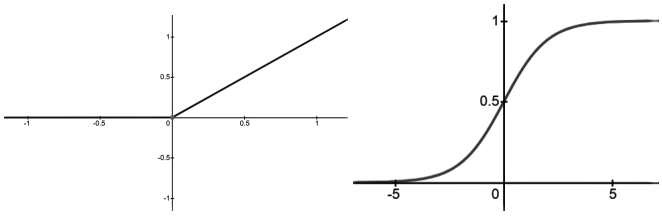

Figure 2.6: *ReLu* and sigmoid activation function

The choice of the activation function is dependent on several aspects but not relevant for this report. The generalized form of feeding forward is then

$$x^{k+1} = \sigma(W^k x^k + b^k),\ 0 \leq k < K \tag{2.2}$$

and the final vector x^K then represents the output. In the case of recognizing handwritten numbers, x^0 has 784 elements, and x^K has 10 elements. The number recognized is n with x^K_{n+1} the highest value of the elements of x^K.

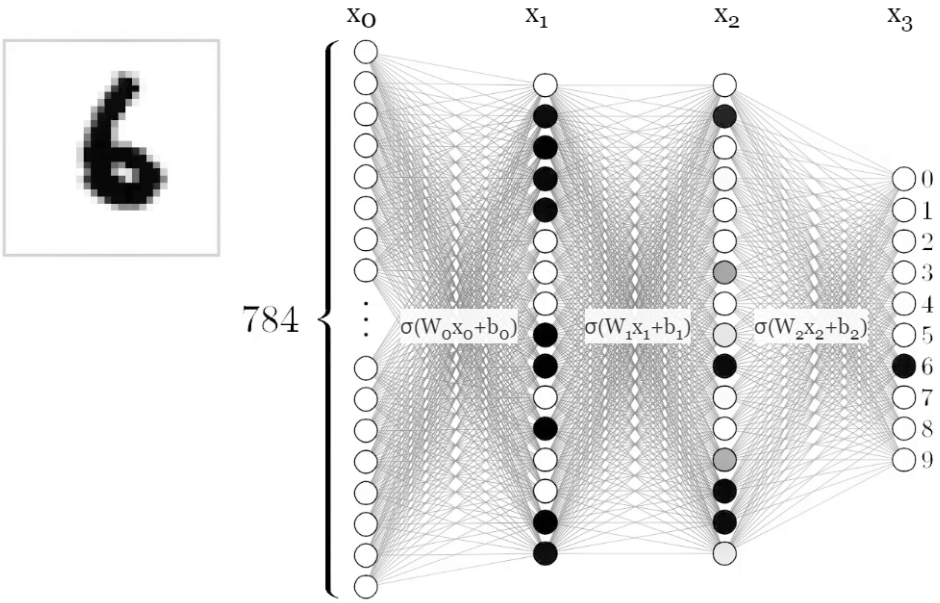

Figure 2.7: Visual representation of MNIST neural network. In layer x_3 the sixth neuron has the highest activation, so the network accurately recognizes the number as a six.

To find the weights and biases the network can be **trained** using images of numbers that have been labeled with the correct number. Start with the weights and biases randomly chosen and run one of these images through the network, which is called **feeding it forward**. Compare the desired output with the calculated output using a **cost function**, adjust the weights and biases accordingly and work backwards through the network. This process is called **back propagation** and is based on gradient descent. The training of the network is not relevant however for this report and will not be discussed further. For more information about back propgation consider reading the book *Deep Learning* by Ian Goodfellow, Yoshua Bengio and Aaron Courville from 2016 [6].

2.1.2. Caveats of DNNs

In addition to explaining DNNs it is also essential to mention their vulnerabilities because the comprehension of these issues is necessary to understand the reason for the line of research followed in this report.

- The number K and the size of the middle layers (defined by the number of elements of x^k and also known as the number of neurons) can be chosen arbitrarily, but it influences your training, computing time, and accuracy.

- While explaining the layers of the network as recognizing loops and sticks can be helpful to understand what the network is doing, a DNN does not actually do this in practice. This can make debugging a DNN difficult because it is hard to understand what exactly is happening between the layers. A fitting term for the middle layers is therefore **hidden layers**.
- DNNs are a form of machine learning, which means they need a dataset to train on. The size of the dataset directly influences the accuracy of the model, which means that DNNs can only be effectively utilized in tandem with large datasets.
- DNNs can be quite vulnerable to faulty inputs. A network with 92% accuracy on a test set can recognize a completely unrecognizable image with the same confidence as an image from the test set. The topic to gain knowledge from adversarial images is called **adversarial learning**. **Adversarial Images** are images that trick the DNN into giving a wrong recognition, while being properly recognizable for humans.

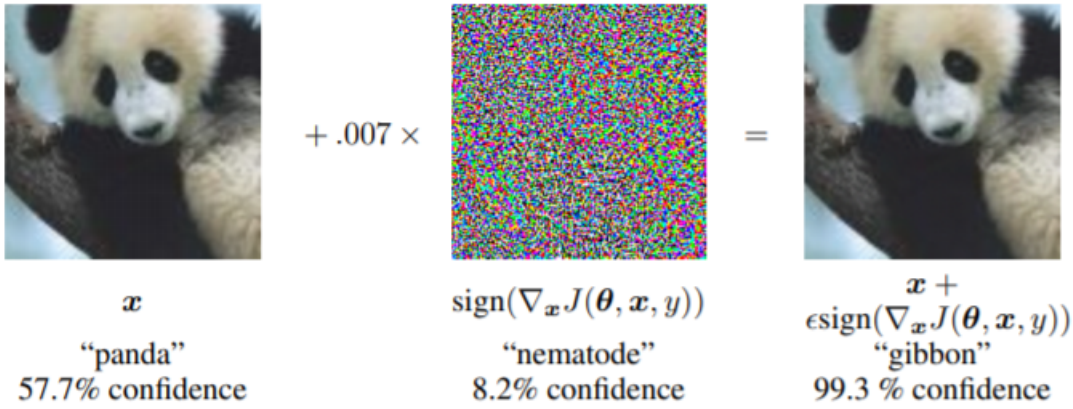

Figure 2.8: Adversarial image of a panda that was recognized by a DNN as a gibbon with 99.3% confidence, having recognized the panda previously with only 57.7% confidence [8]. An almost seemingly random image that is recognized by the DNN as a nematode is added to the original panda image with factor 0.007 to make the adversarial image.

2.2. Mixed Integer Linear Program

A Linear Program is a problem where the goal is to maximize or minimize an objective function under certain linear restrictions. While the study of Linear Programming is a very broad subject with a huge number of applications a small example shall be used to illustrate what it can be used for. The extension from Linear Programs (hereafter referred to as LPs) to Mixed Integer Linear Programs is then easily made.
Consider a student looking to make some extra money to lighten their student debt by planting vegetables in their garden and selling them. The garden has a size of $12m^2$ and they have access to two kinds of vegetables: tomatoes and cucumbers. Tomatoes bring in 18 euros per square meter, and the cucumbers 12. The tomatoes take up 5 hours of labour per square meter, while the cucumbers take up only 3 hours and because they also need to finish their bachelor's report, they only have 40 hours to work on the garden per month. The following problem is then drawn up as an LP:

$$\begin{aligned} &\text{Maximize } 18x_t + 12x_c \\ &\text{subject to } x_t + x_c \leq 12, \\ &\qquad 5x_t + 3x_c \leq 40, \\ &\qquad x_t, x_c \geq 0 \end{aligned} \tag{2.3}$$

with x_t and x_c the size of the areas with t for tomatoes and c for cucumbers. We refer to the part that needs to be minimized or maximized as the **objective function**. The variables that the objective function depends on are called **decision variables** and the rules set upon them are fittingly called **restrictions**. It can be mentioned that the restriction that $x_t, x_c \geq 0$ is implied by the nature of the problem; garden patios with a negative size have yet to be invented in this universe. This restriction is not compulsory for an LP and LPs can also be defined for negative values. In LPs the process of finding good feasible solutions quickly is called **heuristics** and can be very useful. Heuristics do not guarantee the optimal solution, but they search for a reasonably

good solution. Optimization does guarantee an optimal solution if it exists.
A very useful property of LPs is that the restrictions form a convex polyhedron and the optimal solution is always found in an extreme point or vertex of this shape (in case that such a solution exists). The convex polyhedron of 2.3 is given by

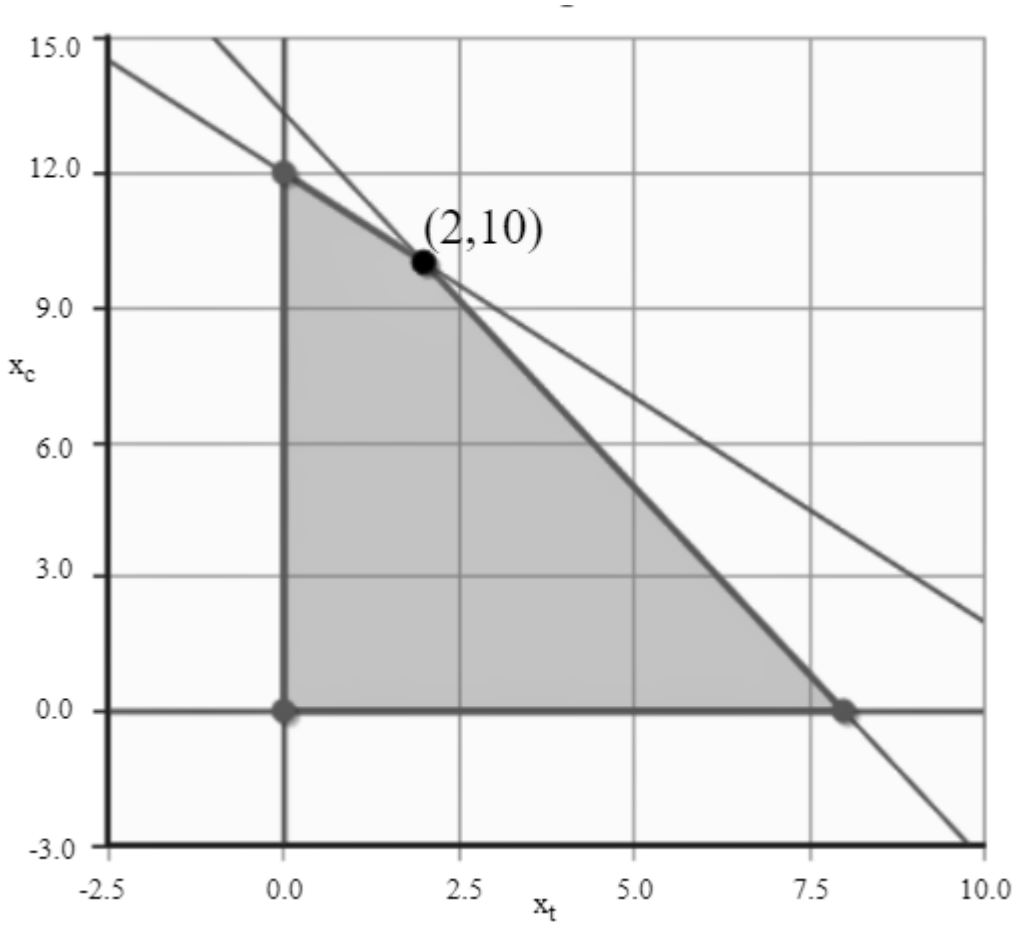

Figure 2.9: Polyhedron from 2.3, the optimal solution is marked with a black dot. The student can make $2 \cdot 18 + 10 \cdot 12 = 156$ euros per month to lighten their debt.

A problem like this can be quickly solved in practice using the simplex algorithm, but it does not need to be mentioned that these problems can easily take daunting sizes with more decision variables and a larger number of restrictions. In this example one could assert that x_c and x_t should be integers, and in this particular example the optimal solution already consists of integer coordinates (but this is usually not the case). When at least one decision variable is restricted to integer values we call a Linear Program a **Mixed Integer Linear Program** (hereafter referred to as MILP).
To give some idea about what makes a MILP more difficult to solve than an LP consider the property of an optimal solution of an LP always being located at a vertex or extreme point of its polyhedron. Now consider the case when all of the variables need to be integer values. The intersection of the polyhedron and the optimal solution does not have to consist of integer coordinates however, like it did in this example. So the actual best solution falls somewhere inside of the polyhedron, and trying all possible solutions is impossible in practice when faced. Especially in case of a MILP, where some values are integers and some are continuous, finding the best solution is a difficult problem: it is NP-Hard unlike LP problems which have polynomial time complexity.

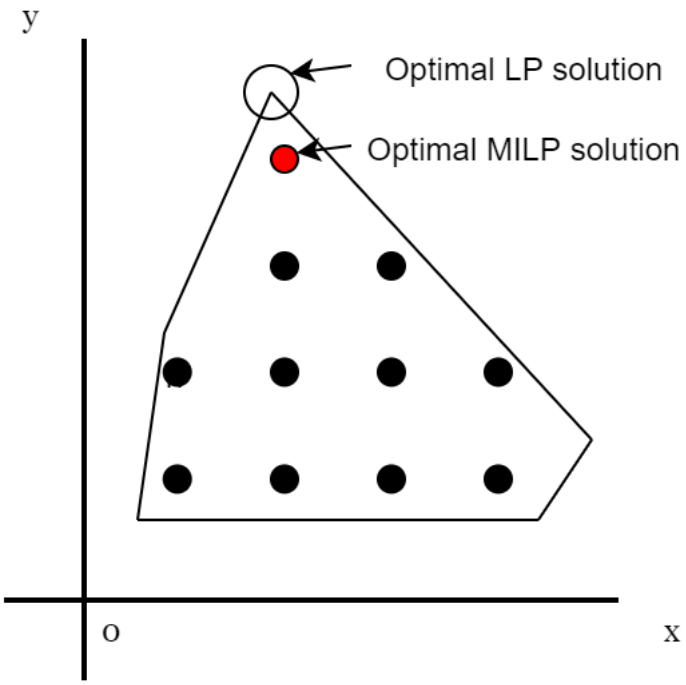

Figure 2.10: A graph of a MILP. Note how the optimal LP solution lies on a non-integer coordinate, and the optimal solution of the MILP problem is therefore in the interior of the polyhedron.

2.2.1. Caveats of MILPs

To understand the difficulties with combining DNNs and MILPs it is necessary to also mention the problems with MILPs.

- Solving a MILP is not a trivial feat, which is why high-quality solvers are usually not free for usage, with some solvers having a free version for students. While the basics of deep learning can be easily learned by someone with little coding experience this does therefore not hold for MILPs.
- The quality of the solver used directly influences the runtime of the problem and because MILPs are NP-hard the computation time can be quite long.
- The runtime of MILPs is also dependent on the tightness posed on the decision variables and finding bounds is often not an easy task.

3

The model

With the prerequisite knowledge of the previous chapter it is now possible to explain the following model. This model was first presented in a 2017 paper by Matteo Fischetti and Jason Jo [5]. In this paper, Fischetti and Jo propose using the weights and biases from an already trained DNN and formulating the weights and biases as an MILP. To make this model more accessible a few minor details will be added to the explanation. This paper has been further discussed in a paper about the robustness of neural networks by Vincent Tjeng and Russ Tedrake [15].

3.1. Formulation

Consider a DNN of $K+1$ layers of neurons, numbered from 0 to K. Layer 0 corresponds to the input layer. The last layer K corresponds to the output of the DNN. The size or amount of neurons of each layer k will be denoted by n_k. The 'transferral' of x_0 through the DNN is defined by

$$x^k = ReLU(W^{k-1}x^{k-1} + b^{k-1}),\ k = 1, ..., K \tag{3.1}$$

where the *ReLU* is the activation function and stands for the rectified linear unit. This function is given by:

$$ReLU : \mathbb{R} \to \mathbb{R}; x \to \max(0, x) \tag{3.2}$$

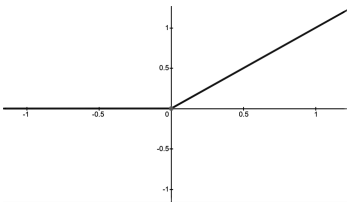

Figure 3.1: Graph of the *ReLu* function.

The *ReLU* is often used as an activation function because it does not have a vanishing gradient unlike the sigmoid function for example. This is an important characteristic when used for training a DNN but in this case it is more relevant because it is a piecewise linear function: a non-linear function would not work in a MILP because the constraints of the MILP would not be linear anymore. Using the *ReLU* as an activation function has several more interesting implications which have been researched in a 2018 paper by Raman Arora, Amitabh Basu, Poorya Mianjy, Anirbit Mukherjee [4]. In this paper the following theorems are put forward:

Theorem 1. *For every natural number N there exists a family of $\mathbb{R} \to \mathbb{R}$ functions such that for any function f in the family we have:*

1. *f is in ReLU-DNN(N^2, N^3)*

2. *f is not in ReLU-DNN(N, $\frac{1}{2}N^n - 1$)*

Theorem 2. *Any ReLU-DNN with n inputs implements a continuous piecewise affine function on $\mathbb{R}^n$. Conversely, any continuous piecewise affine function on $\mathbb{R}^n$ can be implemented by some ReLU-DNN. Moreover, at most $log(n+1)$ hidden layers are needed.*

Here the notation $ReLU - DNN(a,b)$ refers to a *ReLU*-trained DNN with a hidden layers of size b. These results are not needed further in this report but were interesting and noteworthy for background information. To actually model the *ReLU* as a linear function the variable s for negative values of x is introduced and the following equation is obtained:

$$W_{j,*}^{k-1} x^{k-1} + b_j^{k-1} = x_j^k - s_j^k,\ x_j^k \geq 0,\ s_j^k \geq 0, 1 \leq j \leq n_k, 1 \leq k \leq K \tag{3.3}$$

which models the process of feeding forward through the DNN. To further correctly model the *ReLU*, Fischetti and Jo pose that either x_j^k or s_j^k must be zero. To see how this works, simply consider two cases: in case one x_j^k is positive, the *ReLU* sends it to itself by definition of 3.2 and therefore s_j^k must be zero to fit equation 3.3 when 3.1 is substituted. Now consider the case when equation 3.3 has a negative solution. The ReLU would send x_j^k to zero. s_j^k then has a positive value to get the negative result to the equation. This might not be the only way to get the same result but it is simple and 3.3 could otherwise have an infinite amount of solutions. To then actually force that either x or s is zero it is possible to add the restriction that $xs = 0$, but this would alter the linear nature of the model. To cope with this in another way Fischetti and Jo introduce a binary activation variable z which is defined such that:

$$\left.\begin{aligned} z = 1 \rightarrow x \leq 0 \\ z = 0 \rightarrow s \leq 0 \\ z \in \{0,1\} \end{aligned}\right\} \tag{3.4}$$

and introduces bounds M^+ and M^- so that x is 'unbounded' when s is zero and vice versa. The values of both M^+ and M^- need to be chosen carefully; if these restrictions are too tight the model will not have a solution and if they are not tight enough the model require a long time to reach a solution. To combine these elements the following model is introduced:

$$\min \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=1}^{K} \sum_{j=1}^{n_k} \gamma_j^k z_j^k \tag{3.5}$$

$$\left.\begin{aligned} \textstyle\sum_{i=1}^{n_{k-1}} W_{ji}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - s_j^k \\ x_j^k, s_j^k \geq 0 \\ z_j^k \in \{0,1\} \\ z_j^k = 1 \rightarrow x_j^k \leq 0 \\ z_j^k = 0 \rightarrow s_j^k \leq 0 \end{aligned}\right\} k = 1,...,K,\ j = 1,...,\ n_k \tag{3.6}$$

$$lb_j^0 \leq x_j^0 \leq ub_j^0,\ j = 1,...,n_0 \tag{3.7}$$

$$\left.\begin{aligned} lb_j^k \leq x_j^k \leq ub_j^k \\ \overline{lb}_j^k \leq s_j^k \leq \overline{ub}_j^k \end{aligned}\right\} k = 1,...K,\ j = 1,...,n_k \tag{3.8}$$

The restrictions over M^+ and M^- were not explicitly mentioned in the original paper but will be formulated here to make the model more easy to reproduce for students with less experience in solving MILP's.

$$\left.\begin{aligned} x_j^k \leq M^+(1 - z_j^k) \\ s_j^k \leq M^- z_j^k \end{aligned}\right\} k = 1,...,K,\ j = 1,...,n_k \tag{3.9}$$

$$-M^- \leq \sum_{i}^{n_{k-1}} W_{ji} x_i^{k-1} + b_j^{k-1} \leq M^+ \tag{3.10}$$

In the case that z_j^k is zero, s_j^k becomes zero as well and x_j^k becomes bounded only by M^+. This means that choosing M^+ very large is effective at simulating x_j^k to be unbounded, but also bad for the runtime of the

program. The opposite happens when z_j^k is one for s_j^k and M^-.
A small addition is needed for the first equality of 3.6 and 3.10: the paper of Fischetti and Jo wrote W_{ij}, but with the definition of W in 3.1 this is impossible. The indexing W_{ji} is more in line with the definition and produces the desired results of the model.
Another note for choosing the bounds on x and s: 3.6 already enforces that lb and $\overline{lb}$ must be positive for $k = 1, ..., K$. For layer 0 the bounds are unnecessary because x^0 represents the input layer which is often already bounded by the definition.

3.2. Proof of new condition

It is important to mention that the weights and biases need to be trained with the *ReLU* as an activation function. This model will otherwise be infeasible for such a DNN. This restriction has not been explored in the original paper and therefore there will be a brief proof given to illustrate this.

Theorem 3. *For the MILP formulation of a DNN put forward by Fischetti and Jo the DNN needs to be trained using the ReLU as the activation function.*

Proof. Let k be any layer in an arbitrary DNN that is not the last layer. Take the neurons x^k and feed them forward in the DNN using a random activation function σ that is well defined for our problem. Proof by contradiction will demonstrate that σ cannot be chosen arbitrarily and the even stronger assertion that it can only be the *ReLU*.
We know that

$$x^k = \sigma(W^{k-1}x^{k-1} + b^{k-1}), \tag{3.11}$$

With x^k the neurons of the kth layer and W and b the weights and biases, the value of these trained by back-propagation. We now attempt to solve this DNN as an MILP and need to find a solution for the the equation

$$W^{k-1}x^{k-1} + b^{k-1} = x^k - s^k \tag{3.12}$$

The model poses the restriction that x must be a positive number. Consider the case that $x^{k+1} = 0$. Then

$$W^{k-1}x^{k-1} + b^{k-1} = -s^k \tag{3.13}$$

and therefore substitution gives

$$\sigma(-s^k) = 0. \tag{3.14}$$

Because s is also positive this means that σ is the zero function for all non-positive values. This means that σ cannot be an arbitrary function. Now consider the case when $x^{k+1} > 0$. Then by definition s is zero and if 3.12 is substituted into 3.11 the following is obtained:

$$x^k = \sigma(x^k), \tag{3.15}$$

and this means that the σ function is defined by

$$\sigma : \mathbb{R} \to \mathbb{R}; x \to \max(0, x) \tag{3.16}$$

Which implies that σ is the *ReLU* function. Proof by contradiction shows that a DNN can only be used for the model if it was trained using the *ReLU* as its activation function. □

3.3. Worked example

Consider the MNIST dataset [11] referred to in the prerequisites chapter. A network with two hidden layers of 64 neurons each is trained on this data and achieves a 93% accuracy. I would like to mention that I wrote the code for the training algorithm from scratch by myself and did not use libraries such as Keras [2] or PyTorch [3]. This means that the runtimes would likely be shorter with a developed library, but to get a good representation of all processes I used this. It also taught me a lot about the DNNs that I otherwise would not have learned. The code can be found on my github account [13]. It is relevant to mention that it is necessary to train the network with the *ReLU* as an activation function because the model is otherwise infeasible, like what was proved in 3.2. The algorithm behind training the network is very interesting, but sadly not relevant for this project which is why it will not be elaborated upon. For the reader who knows the terminology: I used

regular back propagation with batch training and some momentum.
By definition, the x^0 vector has 784 elements and all values are between 0 and 1. Two hidden layers means four layers of neurons in total, so $K = 3$. Take an image that was correctly recognized by the DNN through the MILP and observe that it is recognized correctly in the MILP as well. Because a method to approximate values for M^+ and M^- has not been discussed yet, set them to 100, so it is certain that the model is feasible with bounds this wide. For the cost functions c and γ choose the zero functions: in case of x^0 being fixed there is only one solution; the one that came out of feeding the input forward through the DNN. This means that they can be chosen arbitrarily so they are zero for simplicity. Lastly take 0 for the lower bounds on all the variables and 5 for the upper bounds. For these estimations I used some trial and error and I have not had a case where they were not wide enough. For solving the model one can use any solver, but for this project the Gurobi [1] package was used in combination with Python. Gurobi is free for students and has a good reputation which is why I preferred it over other solvers. It also comes with a Python library which makes it easy to use.
Use an image that was properly recognized by the DNN and entered into the MILP. It produced the right result on my computer within 3.5 seconds.

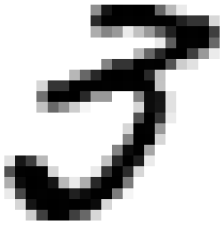

Table 3.1: figure
Image from the MNIST dataset [11], correctly recognized by the DNN.

Output vector	value
x_1^3	0.0
x_2^3	0.0
x_3^3	0.0
x_4^3	≈ 0.51
x_5^3	0.0
x_6^3	≈ 0.12
x_7^3	0.0
x_8^3	0.0
x_9^3	0.0
x_{10}^3	0.0

Table 3.2: Output of the MILP. This is the same as the output vector when the image is fed through the DNN. Not how at the largest value is x_4^3, so the MILP correctly recognized the image as a 3.

Next I ran 100 random images from the dataset and achieved 92 right answers, compliant with the 93 percent accuracy of the DNN. The one difference is attributed to the randomness of selecting the test images. Repeating this experiment provided similar results: almost all MILPs terminated within 5 seconds and the accuracy of the MILP was equivalent to that of the DNN.

4

Adversarial training with the MILP model

Now that the original model is established, it is possible to either find some computational improvement on the model or to find an application to use the model. One of the problems with the MILP is that it needs a trained DNN to work, so it cannot be a replacement for the classic DNN. It is also not possible to train the MILP because you would have to make the weights and biases variable resulting in a Mixed Integer Quadratic Program, which is even more time consuming [12]. Solving the MILP is also much slower than feeding an image forward through the DNN so it can also not be a replacement for that process. To find a use for the MILP it is interesting to look at pitfalls of deep learning and see if the MILP could assist in these problems.
One of the problems with deep learning is that it needs a large dataset to achieve a high accuracy. For some common problems such as handwritten digit or letter recognition there are large repositories of data, but a student with limited resources will have a difficult time if their research proposal needs very specific data points and their only option is generating a dataset themselves. Some data may also just be very difficult to require so it is a worthy line of research to look into the possibilities of a DNN that trains effectively on a small data set.

4.1. Adversarial images from the MILP

One of the applications of the MILP model brought forward by Fischetti and Jo is the ability to generate adversarial images. The adversarials that can be generated are minimal changes to an existing image to have it recognized as a different number than originally recognized. The example put forward by Fischetti and Jo is to have a correctly recognized image recognized wrongly. By the definition of the objective function it is defined as the minimal change.
To generate an adversarial image, take an input vector or for sake of example, image $\tilde{x}$ from the MNIST dataset that has label $\tilde{q}$ and desired adversarial label q. For this application to actually work it is necessary that the DNN used does not recognize the image with label q. The recognition of the image by the DNN can however be different from the actual number pictured, namely the label $\tilde{q}$.
A variable d is introduced that is restricted by:

$$-d_j \leq x_j^0 - \tilde{x}_j^0 \leq d_j,\ d_j \geq 0, j = 1, ..., n_0 \tag{4.1}$$

d acts as a way to measure the difference between the pictures x and $\tilde{x}$. $\tilde{x}$ has been chosen and has fixed values. The goal is to have the new image x look as much like $\tilde{x}$ as possible. This means that the difference between them is bounded by d and d is then forced by the model to be as small as possible.
To then enforce that x is actually recognized as q, the restriction

$$x_{q+1}^K \geq 1.2 x_{j+1}^K,\ j \in \{0, ..., 9\} \setminus q \tag{4.2}$$

needs to be imposed. This means that in the last layer, the activation of neuron x_{q+1}^K in the output layer must be bigger than the value in all other neurons in the output layer. The value 1.2 is suggested by Fischetti and Jo, but it is also possible to have a different number larger or equal to 1 (in case of equal to 1, the $\geq$ sign becomes a $>$). The higher this number the more certain the classification is. In an ideal case the number would approach infinity because the activation of the desired number is 1 and all other activations are zero.

The objective function becomes $\sum_{j=1}^{n_0} d_j$, to minimize the change made to the original picture. The new image then becomes x^0. While this application is interesting, it is worthy mentioning that to calculate an image a different label needs to be chosen. It would be an interesting question how one would calculate the smallest change to an image to have it not be recognized correctly anymore.

This variation on the model took a lot more time on my computer though, most images had a solution within 10 seconds but some took almost a minute to calculate.

Figure 4.1: On the left the original image before calculating adversarial, on the right the adversarial image.

Number	Activation original	Activation adversarial
0	0	0
1	0	0
2	0	0
3	≈ 0.05	≈ 0.04
4	≈ 0.21	≈ 0.42
5	0	0
6	0	0
7	≈ 0.07	≈ 0.15
8	0	0
9	≈ 0.53	≈ 0.35

Figure 4.2: Activations of the two images for every digit. Image was recognized before as a 9 and now as a 4.

An interesting variation on this change to the model is to take a wrongly recognized image and calculate the smallest change it needs to be recognized correctly. This would be enforced by changing the desired label q to be the same as $\tilde{q}$. This only works when the image has been wrongly labeled by the DNN, otherwise the adversarial generated is the original image because there is no change needed to have it recognized as the right label.

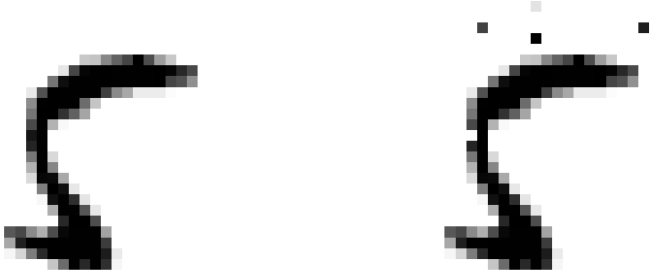

Figure 4.3: Original image on the left, adversarial on the right.

Number	Activation original	Activation adversarial
0	≈ 0.37	0
1	0	0
2	0	0
3	0	0
4	0	0
5	≈ 0.06	$\approx 5.55 \cdot 10^{-17}$
6	0	0
7	0	0
8	≈ 0.12	0
9	0	0

Figure 4.4: Activations of the two images for every digit. Notice how the image was recognized as a 0 with some confidence and five well placed dots made the image completely unrecognizable.

From the start of this project one question stood out to me: what would happen if you would take the adversarial images generated by the MILP and trained the DNN on these images once more. Would this influence the accuracy of the network? To the best of my knowledge this question does not have a clear answer and has not been investigated previously in this setup.

4.2. Experiment setup

Because the training of the DNN has some amount of randomness (it is appropriate to start with random weights and zero biases) I ran the same experiments 20 times and observed the average of the accuracy every epoch. One epoch means that the network has gone over every image in the dataset only once.
The experiment that I am running considers a DNN with one hidden layer of 64 neurons. The dataset we are using is the same as used before, namely the MNIST dataset [11]. I drew 100 images from the dataset semi-randomly: because of the small sample size I took 10 images of every number randomly so the network would be able to see every number an equal amount of times. I trained the network for 9 epochs. I ran a control group without any aid of an MILP to compare the results of the MILP with. The experiment I tried was taking all the images that were correctly recognized by the DNN, use the MILP to calculate their minimal adversary in comparison to the number that it is often confused by. Usually the nine is confused for a four, three for an eight, et cetera. I then added these images to the training set and repeated the training like usual, removing the adversarials from the set after one epoch to prevent the DNN from overfitting. Doing this for every epoch proved to be too time-consuming: doing it for just one epoch took about 8 hours to run the experiment 20 times. For reference, the 20 networks trained without any use of the MILP took about half an hour to compute. I therefore consider four cases: no adversarial training, adding adversarial images the second epoch, the fifth epoch, or the ninth epoch.

4.3. Results

The results produced were surprising to say the least. The averages of the accuracy per epoch were nearly identical, seeing no significant in- or decrease at the point of adversarial training. This is the polar opposite of the expected result.

	None	first epoch	ninth epoch
epoch 1	46%	48%	41%
epoch 2	79%	78%	76%
epoch 3	86%	88%	87%
epoch 4	92%	92%	92%
epoch 5	95%	95%	94%
epoch 6	96%	97%	96%
epoch 7	98%	98%	97%
epoch 8	98%	99%	97%
epoch 9	99%	99%	99%

Figure 4.5: Average training accuracy of 20 networks over 100 samples

	None	first epoch	ninth epoch
epoch 1	35%	37%	29%
epoch 2	55%	55%	54%
epoch 3	61%	62%	61%
epoch 4	65%	65%	65%
epoch 5	67%	66%	67%
epoch 6	68%	68%	69%
epoch 7	69%	68%	70%
epoch 8	70%	69%	71%
epoch 9	71%	70%	71%

Figure 4.6: Average testing accuracy of 20 networks over 69900 samples

The small differences are likely to be attributed to the randomness of the setup and do not constitute a valid result. The method used to train the network was back-propgation and it could be tested to see if it provides similar results.

4.4. Implications

While the adversarial training has not proven itself useful for this setup, there is something to be said about the results that is remarkable. Machine learning is centered around having a large dataset and having a computer train on it to predict the identity of not before seen data points. The philosophy behind this is that is quite similar to how humans learn: you teach a child what is a cat by showing it a cat and when they point to a dog and say 'cat!' we correct them and tell them it's not a cat, it's a dog. This approach is very effective but also seems to point people to the conclusion that in the end computers and humans think alike, to the point where news outlets will seem to almost speak about DNNs as if they are living organisms [10].
In that way of thinking, surely training a network with the adversarial images with minimal change should be effective, but the experiment produced no result. This could have to do with the identity of the adversarials and maybe a different adversarial could be effective, but it could also mean that the computer just does not 'see' the image in the way we like to explain that it does. In the end a human sees an image, and a computer a sequence of greyscale pixel values between 0 and 1. To see how the DNN actually sees the image, the MILP can be easily modified to generate a picture that maximizes the activation of a certain number. This is done by making the input layer variable and having the objective function of the MILP be the cost function of the DNN.

Figure 4.7: A ‘perfect’ 8 according to the DNN. All digits except 8 have activation 0 and 8 has activation 1.

We see that this is nothing like an actual number, but it is what the DNN considers to be a perfect image. The DNN did not prove itself useful in the experiment from above, but it does show a lot of interesting properties of the DNN.

5
Conclusion

In this report we looked at a crossover between Deep learning and Optimization. These fields are usually not taught together so there are not a lot of researchers looking at combining the two fields. We have seen however that there is potential in analyzing DNNs and it is exciting to await further research. We looked at a model that can be used for generating adversarial images and it is easy to see that DNNs surely are vulnerable if a well placed dot can confuse the network quite easily. These images have not proven themselves useful in this setup for adversarial training which is a shame but nonetheless interesting: the thought of training the network on these adversarial images seems like a setup that would work. The way that DNNs 'think' is therefore probably not as similar to humans as the media likes to portray it [10].
In addition to looking at adversarial training we have also seen an added constraint to the model that had not been stated before but is an important addition. Also a variation on the adversarial that in stead of duping the image to a close look-a-like we have seen the minimal change needed to have a wrongly recognized image be recognized correctly.
I hope that seeing the adversarial images makes you just as excited as I was the first time I saw them and that you enjoyed reading this report as much I enjoyed doing this project. Whether you are involved with optimization or machine learning, I hope that you became excited about the other field and are also excited to see what more research about this promising tag-team can bring to both fields.

5.1. Further research

Speaking about further research, during this project there is so much more I would have liked to do but this bachelor student needs to get back to her debt-relieving vegetable garden. I will state these ideas and hope somebody else will look at them in the future.

5.1.1. Other input formats

During this project I only worked with the MNIST dataset [11]. It would be interesting to see what would happen if you would combine the model with other image datasets or even sound or video datasets. The input layer would have a lot more elements however so compuational improvements would be

5.1.2. Other activation functions

At the start of the project I had difficulty at implementing the MILP model, because I had used the sigmoid function to train my network. This is how I found that this is an important constraint to the model. Because the *ReLU* is not the only activation function to train DNNs it would be interesting if the model could be extended to other activation functions.

5.1.3. Generative Adversarial Networks

Generative Adversarials Networks (hereafter referred to as GANs) is a combination of two DNNs that work on one problem together. It is quite a new field of machine learning, being first proposed in 2014 [7] and has a lot of promise. The first DNN creates an image or other format and sends it to the second DNN. The second DNN then judges if the image came from the training dataset of the first DNN or if it had generated itself. The

DNNs then play a sort of pingpong match to make the images more and more convincing. It would be really interesting to see what would happen if the MILP were to be used to create adversarials.

Figure 5.1: An image generated by StyleGAN, a GAN developed by Nvidia researchers [9]. The image looks like a real photo but is fully computer generated.

Bibliography

[1] Gurobi: the fastest solver. https://gurobi.com/. Accessed: 2020-01-06.

[2] Keras: The python deep learning library. https://keras.io/. Accessed: 2020-01-06.

[3] Pytorch: from research to production. https://pytorch.org/. Accessed: 2020-01-06.

[4] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *CoRR*, abs/1611.01491, 2016. URL http://arxiv.org/abs/1611.01491.

[5] Matteo Fischetti and Jason Jo. Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. *CoRR*, abs/1712.06174, 2017. URL http://arxiv.org/abs/1712.06174.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *CoRR*, abs/1406.2661, 2014. URL https://arxiv.org/abs/1406.2661.

[8] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014. URL https://arxiv.org/abs/1412.6572.

[9] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018. URL http://arxiv.org/abs/1812.04948.

[10] Ivo Landman. Van de fictie van orwell tot brexit-nepnieuws, tekstrobot gpt-2 kan alles. *NOS*. URL https://nos.nl/artikel/2276233-van-de-fictie-van-orwell-tot-brexit-nepnieuws-tekstrobot-gpt-2-kan-alles.html.

[11] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs*, 2, 2010. URL http://yann.lecun.com/exdb/mnist.

[12] Alberto Del Pia, Santanu S. Dey, and Marco Molinaro. Mixed-integer quadratic programming is in NP. *CoRR*, abs/1407.4798, 2014. URL http://arxiv.org/abs/1407.4798.

[13] M.A.E. Schönfeld. Dnn milp. https://github.com/marietteschonfeld/DNN_MILP, 2020.

[14] Shai Shalev-Shwartz and Shai Ben-David. *Understand machine learning*. Cambridge University Press, 2014.

[15] Vincent Tjeng and Russ Tedrake. Verifying neural networks with mixed integer programming. *CoRR*, abs/1711.07356, 2017. URL http://arxiv.org/abs/1711.07356.

[16] Laurence A. Wolsey. *Integer programming*. John Wiley Sons Inc, 1998.

A

Test results

These are the results I achieved of running the same experiment twenty times with small changes. The results are the accuracy on the training set and testing set. The adversarial images were not considered for the accuracy because the goal of the experiment was to see if training on the adversarial images provided a higher accuracy on the testing set.

Epoch 1		Epoch 2		Epoch 3		Epoch 4		Epoch 5		Epoch 6		Epoch 7		Epoch 8		Epoch 9	
Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
35.00%	31.40%	79.00%	55.87%	83.00%	59.47%	87.00%	65.03%	92.00%	67.86%	97.00%	70.37%	96.00%	68.80%	98.00%	71.19%	99.00%	71.51%
44.00%	34.53%	84.00%	57.23%	92.00%	63.63%	93.00%	66.46%	94.00%	67.93%	95.00%	71.31%	97.00%	69.77%	98.00%	73.21%	99.00%	74.05%
46.00%	36.76%	77.00%	55.27%	85.00%	61.72%	92.00%	65.67%	95.00%	68.34%	96.00%	69.42%	98.00%	70.93%	97.00%	69.37%	98.00%	71.81%
54.00%	35.71%	78.00%	58.24%	85.00%	59.96%	93.00%	66.67%	97.00%	69.17%	97.00%	68.65%	98.00%	68.85%	98.00%	70.19%	99.00%	70.68%
53.00%	35.68%	71.00%	50.35%	86.00%	61.71%	90.00%	65.23%	95.00%	67.14%	97.00%	68.11%	99.00%	70.24%	99.00%	70.46%	99.00%	70.88%
45.00%	33.13%	79.00%	52.24%	86.00%	59.54%	88.00%	63.56%	98.00%	67.13%	96.00%	67.68%	98.00%	68.44%	98.00%	69.02%	99.00%	70.46%
52.00%	39.88%	82.00%	59.77%	88.00%	64.07%	95.00%	69.28%	96.00%	69.26%	97.00%	70.84%	97.00%	71.49%	97.00%	70.33%	99.00%	72.02%
50.00%	36.58%	82.00%	54.88%	84.00%	60.58%	94.00%	66.25%	97.00%	67.44%	97.00%	66.71%	98.00%	69.91%	99.00%	71.33%	99.00%	71.17%
48.00%	34.92%	68.00%	50.13%	88.00%	61.15%	98.00%	64.30%	98.00%	67.52%	98.00%	67.24%	99.00%	69.52%	100.00%	70.18%	100.00%	70.77%
44.00%	30.37%	78.00%	54.74%	88.00%	60.20%	90.00%	64.62%	93.00%	67.62%	97.00%	70.38%	95.00%	69.67%	98.00%	69.57%	99.00%	71.40%
42.00%	33.36%	82.00%	57.77%	87.00%	61.87%	91.00%	65.28%	92.00%	65.60%	93.00%	67.40%	97.00%	70.23%	98.00%	70.65%	99.00%	70.71%
46.00%	36.97%	78.00%	57.96%	80.00%	57.03%	88.00%	65.28%	93.00%	67.41%	96.00%	68.37%	97.00%	70.84%	98.00%	72.37%	98.00%	70.93%
42.00%	30.96%	76.00%	53.72%	85.00%	61.54%	89.00%	64.68%	95.00%	67.80%	98.00%	68.25%	99.00%	69.06%	99.00%	69.30%	99.00%	69.91%
46.00%	37.46%	82.00%	57.04%	81.00%	61.10%	90.00%	63.93%	94.00%	67.42%	93.00%	66.42%	96.00%	68.79%	99.00%	70.17%	98.00%	69.53%
49.00%	37.18%	80.00%	52.90%	95.00%	61.57%	92.00%	62.27%	96.00%	64.99%	99.00%	67.14%	96.00%	67.57%	99.00%	68.66%	99.00%	69.29%
45.00%	31.59%	77.00%	55.99%	87.00%	61.99%	92.00%	66.38%	94.00%	67.40%	97.00%	68.74%	98.00%	68.99%	97.00%	68.71%	99.00%	70.47%
53.00%	38.89%	74.00%	55.59%	87.00%	62.19%	94.00%	66.07%	96.00%	68.14%	95.00%	67.96%	99.00%	67.17%	98.00%	69.68%	99.00%	69.64%
44.00%	34.47%	80.00%	55.06%	91.00%	60.94%	94.00%	65.75%	95.00%	67.11%	97.00%	67.74%	99.00%	68.91%	98.00%	70.72%	99.00%	70.21%
48.00%	33.79%	86.00%	59.15%	84.00%	61.87%	93.00%	65.16%	97.00%	68.24%	96.00%	68.66%	97.00%	69.97%	96.00%	66.96%	99.00%	69.30%
43.00%	31.27%	75.00%	51.92%	89.00%	56.06%	89.00%	60.81%	95.00%	61.90%	97.00%	64.15%	97.00%	64.86%	98.00%	66.65%	99.00%	67.17%
34.00%	27.39%	78.00%	55.38%	84.00%	61.73%	95.00%	62.82%	98.00%	66.86%	98.00%	67.93%	99.00%	68.51%	99.00%	69.33%	99.00%	69.63%
45.00%	36.90%	81.00%	58.43%	87.00%	61.36%	88.00%	64.48%	91.00%	66.84%	96.00%	69.54%	96.00%	69.94%	96.00%	70.87%	98.00%	71.81%

Figure A.1: Test results of training without adversarial images.

Epoch 1		Epoch 2		Epoch 3		Epoch 4		Epoch 5		Epoch 6		Epoch 7		Epoch 8		Epoch 9	
Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
40.00%	37.00%	70.00%	52.60%	84.00%	60.51%	84.00%	62.56%	92.00%	65.67%	93.00%	67.03%	98.00%	69.21%	97.00%	70.29%	99.00%	70.75%
40.00%	31.01%	74.00%	51.47%	88.00%	58.34%	90.00%	59.77%	89.00%	61.67%	93.00%	65.06%	94.00%	64.60%	97.00%	67.28%	97.00%	67.56%
43.00%	32.46%	76.00%	55.23%	85.00%	62.04%	91.00%	63.88%	93.00%	67.01%	94.00%	69.43%	98.00%	69.76%	98.00%	68.71%	99.00%	70.85%
41.00%	33.09%	77.00%	55.70%	89.00%	60.17%	89.00%	63.22%	96.00%	67.75%	97.00%	67.25%	98.00%	70.04%	98.00%	70.50%	98.00%	70.29%
36.00%	28.42%	75.00%	47.81%	87.00%	60.88%	93.00%	60.90%	95.00%	65.77%	96.00%	65.53%	99.00%	70.73%	97.00%	70.03%	100.00%	71.54%
48.00%	37.13%	77.00%	52.72%	86.00%	61.16%	92.00%	64.62%	93.00%	64.40%	93.00%	66.72%	97.00%	67.84%	99.00%	69.61%	99.00%	70.01%
44.00%	35.75%	82.00%	57.95%	89.00%	63.52%	94.00%	67.95%	95.00%	67.01%	96.00%	67.97%	98.00%	69.25%	99.00%	68.14%	99.00%	70.24%
54.00%	40.45%	85.00%	56.62%	94.00%	65.89%	94.00%	68.33%	95.00%	67.72%	97.00%	68.67%	99.00%	68.89%	100.00%	70.03%	100.00%	69.69%
46.00%	35.07%	79.00%	57.03%	79.00%	56.40%	94.00%	65.94%	95.00%	63.15%	98.00%	69.07%	98.00%	67.54%	99.00%	69.67%	99.00%	67.83%
54.00%	40.54%	75.00%	57.63%	86.00%	61.81%	89.00%	63.23%	91.00%	65.90%	99.00%	66.05%	99.00%	67.10%	100.00%	68.35%	99.00%	67.48%
55.00%	43.30%	72.00%	51.72%	89.00%	64.13%	92.00%	63.41%	95.00%	67.56%	94.00%	66.00%	98.00%	63.51%	98.00%	67.68%	98.00%	67.15%
58.00%	42.97%	80.00%	54.97%	95.00%	64.27%	96.00%	68.92%	99.00%	66.85%	100.00%	69.58%	100.00%	70.44%	100.00%	68.90%	100.00%	69.70%
50.00%	40.96%	84.00%	56.94%	94.00%	62.88%	96.00%	67.25%	98.00%	69.09%	97.00%	68.80%	100.00%	70.36%	99.00%	70.70%	100.00%	71.83%
45.00%	37.05%	75.00%	53.48%	89.00%	64.18%	89.00%	64.45%	96.00%	66.32%	98.00%	67.80%	99.00%	69.55%	100.00%	70.16%	100.00%	70.91%
47.00%	37.56%	78.00%	56.81%	90.00%	63.85%	87.00%	64.89%	95.00%	67.24%	98.00%	68.00%	98.00%	65.13%	100.00%	69.17%	100.00%	67.77%
40.00%	29.50%	76.00%	50.05%	84.00%	59.13%	93.00%	64.30%	92.00%	63.13%	94.00%	65.21%	97.00%	65.73%	99.00%	66.33%	99.00%	67.89%
55.00%	41.32%	77.00%	55.37%	86.00%	62.80%	93.00%	62.42%	96.00%	66.11%	100.00%	67.51%	99.00%	69.18%	99.00%	70.05%	100.00%	70.10%
55.00%	37.68%	84.00%	59.22%	88.00%	62.47%	94.00%	70.11%	96.00%	68.37%	97.00%	69.70%	100.00%	71.32%	100.00%	71.66%	100.00%	71.23%
55.00%	40.93%	75.00%	52.86%	87.00%	61.33%	93.00%	63.97%	97.00%	67.13%	98.00%	64.85%	97.00%	67.20%	99.00%	67.58%	100.00%	68.51%
47.00%	39.30%	79.00%	57.72%	89.00%	63.38%	94.00%	68.04%	96.00%	66.93%	98.00%	70.81%	99.00%	69.32%	99.00%	67.70%	100.00%	69.22%
34.00%	27.39%	78.00%	55.38%	84.00%	61.73%	95.00%	62.82%	98.00%	66.86%	98.00%	67.93%	99.00%	68.51%	99.00%	69.33%	99.00%	69.63%
45.00%	36.90%	81.00%	58.43%	87.00%	61.36%	88.00%	64.48%	91.00%	66.84%	96.00%	69.54%	96.00%	69.94%	96.00%	70.87%	98.00%	71.81%

Figure A.2: Test results of training with adversarial images the second epoch.

Epoch 1		Epoch 2		Epoch 3		Epoch 4		Epoch 5		Epoch 6		Epoch 7		Epoch 8		Epoch 9	
Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train
40.00%	37.00%	70.00%	52.60%	84.00%	60.51%	84.00%	62.56%	92.00%	65.67%	93.00%	67.03%	98.00%	69.21%	97.00%	70.29%	99.00%	70.75%
40.00%	31.01%	74.00%	51.47%	88.00%	58.34%	90.00%	59.77%	89.00%	61.67%	93.00%	65.06%	94.00%	64.60%	97.00%	67.28%	97.00%	67.56%
43.00%	32.46%	76.00%	55.23%	85.00%	62.04%	91.00%	63.88%	93.00%	67.01%	94.00%	69.43%	98.00%	69.76%	98.00%	68.71%	99.00%	70.85%
41.00%	33.09%	77.00%	55.70%	89.00%	60.17%	89.00%	63.22%	96.00%	67.75%	97.00%	67.25%	98.00%	70.04%	98.00%	70.50%	98.00%	70.29%
36.00%	28.42%	75.00%	47.81%	87.00%	60.88%	93.00%	60.90%	95.00%	65.77%	96.00%	65.53%	99.00%	70.73%	97.00%	70.03%	100.00%	71.54%
48.00%	37.13%	77.00%	52.72%	86.00%	61.16%	92.00%	64.62%	93.00%	64.40%	93.00%	66.72%	97.00%	67.84%	99.00%	69.61%	99.00%	70.01%
44.00%	35.75%	82.00%	57.95%	89.00%	63.52%	94.00%	67.95%	95.00%	67.01%	96.00%	67.97%	98.00%	69.25%	99.00%	68.14%	99.00%	70.24%
54.00%	40.45%	85.00%	56.62%	94.00%	65.89%	94.00%	68.33%	95.00%	67.72%	97.00%	68.67%	99.00%	68.89%	100.00%	70.03%	100.00%	69.69%
46.00%	35.07%	79.00%	57.03%	79.00%	56.40%	94.00%	65.94%	95.00%	63.15%	98.00%	69.07%	98.00%	67.54%	99.00%	69.67%	99.00%	67.83%
54.00%	40.54%	75.00%	57.63%	86.00%	61.81%	89.00%	63.23%	91.00%	65.90%	99.00%	66.05%	99.00%	67.10%	100.00%	68.35%	99.00%	67.48%
55.00%	43.30%	72.00%	51.72%	89.00%	64.13%	92.00%	63.41%	95.00%	67.56%	94.00%	66.00%	98.00%	63.51%	98.00%	67.68%	98.00%	67.15%
58.00%	42.97%	80.00%	54.97%	95.00%	64.27%	96.00%	68.92%	99.00%	66.85%	100.00%	69.58%	100.00%	70.44%	100.00%	68.90%	100.00%	69.70%
50.00%	40.96%	84.00%	56.94%	94.00%	62.88%	96.00%	67.25%	98.00%	69.09%	97.00%	68.80%	100.00%	70.36%	99.00%	70.70%	100.00%	71.83%
45.00%	37.05%	75.00%	53.48%	89.00%	64.18%	89.00%	64.45%	96.00%	66.32%	98.00%	67.80%	99.00%	69.55%	100.00%	70.16%	100.00%	70.91%
47.00%	37.56%	78.00%	56.81%	90.00%	63.85%	87.00%	64.89%	95.00%	67.24%	98.00%	68.00%	98.00%	65.13%	100.00%	69.17%	100.00%	67.77%
40.00%	29.50%	76.00%	50.05%	84.00%	59.13%	93.00%	64.30%	92.00%	63.13%	94.00%	65.21%	97.00%	65.73%	99.00%	66.33%	99.00%	67.89%
55.00%	41.32%	77.00%	55.37%	86.00%	62.80%	93.00%	62.42%	96.00%	66.11%	100.00%	67.51%	99.00%	69.18%	99.00%	70.05%	100.00%	70.10%
55.00%	37.68%	84.00%	59.22%	88.00%	62.47%	94.00%	70.11%	96.00%	68.37%	97.00%	69.70%	100.00%	71.32%	100.00%	71.66%	100.00%	71.23%
55.00%	40.93%	75.00%	52.86%	87.00%	61.33%	93.00%	63.97%	97.00%	67.13%	98.00%	64.85%	97.00%	67.20%	99.00%	67.58%	100.00%	68.51%
47.00%	39.30%	79.00%	57.72%	89.00%	63.38%	94.00%	68.04%	96.00%	66.93%	98.00%	70.81%	99.00%	69.32%	99.00%	67.70%	100.00%	69.22%
34.00%	27.39%	78.00%	55.38%	84.00%	61.73%	95.00%	62.82%	98.00%	66.86%	98.00%	67.93%	99.00%	68.51%	99.00%	69.33%	99.00%	69.63%
45.00%	36.90%	81.00%	58.43%	87.00%	61.36%	88.00%	64.48%	91.00%	66.84%	96.00%	69.54%	96.00%	69.94%	96.00%	70.87%	98.00%	71.81%

Figure A.3: Test results of training with adversarial images the ninth epoch.