Hochschule Niederrhein
University of Applied Sciences

Elektrotechnik
und Informatik
Faculty of Electrical Engineering
and Computer Science

Training of ReLU Activated Multilayerd Neural Networks with Mixed Integer Linear Programs

Steffen Goebbels

Technischer Bericht Nr. 2021-01

IMPRESSUM

Technische Berichte des Fachbereichs Elektrotechnik und Informatik,
Hochschule Niederrhein

ISSN 2199-031X

HERAUSGEBER

Christoph Dalitz und Steffen Goebbels
Fachbereich Elektrotechnik und Informatik

ANSCHRIFT

Hochschule Niederrhein
Reinarzstr. 49
47805 Krefeld

http://www.hsnr.de/fb03/technische-berichte/

S. Goebbels: „Training of ReLU Activated Multilayerd Neural Networks with Mixed Integer Linear Programs.“
Technischer Bericht Nr. 2021-01, Hochschule Niederrhein, Fachbereich Elektrotechnik und Informatik, 2021

Training of ReLU Activated Multilayer Neural Networks with Mixed Integer Linear Programs

Steffen Goebbels
iPattern Institute, Niederrhein University of Applied Sciences,
Reinarzstr. 49, 47805 Krefeld, Germany
steffen.goebbels@hsnr.de

Abstract

In this paper, it is demonstrated through a case study that multilayer feedforward neural networks activated by ReLU functions can in principle be trained iteratively with Mixed Integer Linear Programs (MILPs) as follows. Weights are determined with batch learning. Multiple iterations are used per batch of training data. In each iteration, the algorithm starts at the output layer and propagates information back to the first hidden layer to adjust the weights using MILPs or Linear Programs. For each layer, the goal is to minimize the difference between its output and the corresponding target output. The target output of the last (output) layer is equal to the ground truth. The target output of a previous layer is defined as the adjusted input of the following layer. For a given layer, weights are computed by solving a MILP. Then, except for the first hidden layer, the input values are also modified with a MILP to better match the layer outputs to their corresponding target outputs. The method was tested and compared with Tensorflow/Keras (Adam optimizer) using two simple networks on the MNIST dataset containing handwritten digits. Accuracies of the same magnitude as with Tensorflow/Keras were achieved.

1 Introduction

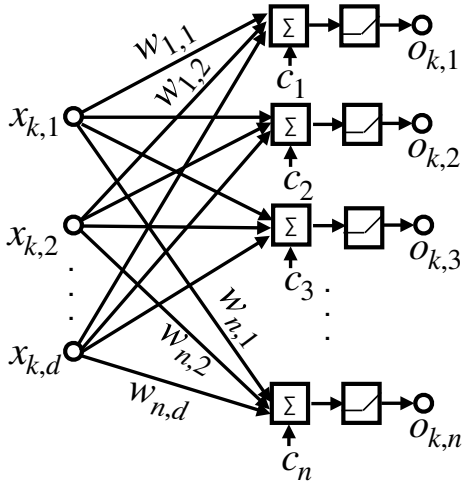
Neural networks typically learn by adjusting weights using nonlinear optimization in a training phase. Variants of gradient descent are often used. These techniques require "some" differentiability of the error functional. Therefore, piecewise linear activation functions like the Rectified Linear Unit (ReLU)

$$\sigma(x) := \max\{0, x\}$$

or the Heaviside function, that are not differentiable at the origin, raise the question of whether linear and mixed integer linear programming techniques are also suitable for network training.

Learning to near optimality can be done with Linear Programs (LP) of exponential size for certain network architectures, see [1]. But this is not applicable in practice. Mixed Integer Linear Programs (MILPs) are proposed in [2] to find inputs of ReLU networks that maximize unit activation. This can help to understand the features computed in the network. To this end, the weights are not variable. The output of a neuron is modeled by the same constraints as in [3], where MILPs are used to count maximum numbers of linear regions in outputs of ReLU networks. In order to find vulnerabilities, a trained binary neural network is attacked by a MILP in [4]. This MILP computes inputs for which the network fails to predict. In both this MILP and in [5], the weights are also considered as constants. Another approach to evaluate the robustness of networks is described in [6]. A network layout consisting of nodes and edges is optimized with a MILP in [7].

The present work investigates the suitability of training with MILPs, i.e. in contrast to the previously mentioned works, network weights are now variables of the optimization problem. Oracle Inc. holds US patent [8], which protects the idea of using MILPs for training parts of (deep) neural networks. The solution described in this patent works in a scenario with piecewise constant activation functions for hidden neurons as well as piecewise linear activation functions on the output layer. Without additional algorithmic intervention, it does not work when values of weights (that are variables in the optimization problem) have to be multiplied. This is the case when activation functions that are not piecewise constant are used on successive layers. Thus, additional considerations are required for ReLU-activated networks to use linear optimization methods.



By applying the ReLU function σ to each component of the vector $W\vec{x}+\vec{c}$, the output $\vec{o}$ is obtained ($\vec{x}\in\mathbb{R}^d$, $\vec{o}\in\mathbb{R}^n$, $\vec{c}\in\mathbb{R}^n$, $W\in\mathbb{R}^{n\times d}$):

$$\vec{o}=\sigma(W\vec{x}+\vec{c}).$$

Figure 1: Building block of ReLU-activated feedforward network: blocks can be concatenated to realize a deep network. The output $\vec{o}$ of a layer then becomes the input $\vec{x}$ of the next layer, i.e., the dimensions of subsequent building blocks have to fit. The building blocks do not share weights.

We investigate a backpropagation-like algorithm (see Algorithm 1) to iteratively train a ReLU network with LPs and MILPs. The prerequisite is a neural network with ReLU activation that is a concatenation of building blocks, as shown in Figure 1. All hidden layers and the output layer consist of such a building block. Edges can be removed by setting their weights fixed to zero. In addition, the equality of weights can be specified. This makes it possible, for example, to realize convolutional layers. The input layer only passes values to the building block of the first hidden layer. Most deep neural networks follow this architecture but use a different activation function like softmax on the output layer. For simplicity, we also use ReLU there.

To evaluate the algorithm, we select the MNIST dataset[1], see [9], that consists of 60.000 images (28×28 pixels) of handwritten digits for training and 10.000 digits for testing, in connection with two small example instances of the discussed network. One instance consists of 784-8-8-8-10 neurons on five layers (three hidden fully connected layers), cf. [2, DNN1]. The values of ten output neurons encode the recognized number. Another example is a 49-25-10 network with a convolutional (single feature-map) and a subsequent fully connected layer. To apply the network, the images are downsampled to a size of 7×7 gray values by taking the mean values of 4×4 regions. The size of the convolution kernel is 3×3, all offsets c_j are set to zero. For both network instances, the index of an output neuron whose output is closest to one represents the detected number. The accuracy is the relative number of true detections.

On batches (subsets) of training data (which sadly must be small because of runtimes), we determine the weights using Algorithm 1, that consists of three MILPs. They are specified in the next section. Then the results are compared with those of gradient descent as implemented by the widely used Adam optimizer [10].

2 Mixed Integer Linear Programs and Linear Programs

2.1 Computation of weights

We determine weights $W\in\mathbb{R}^{n\times d}$,

$$W=[w_{l,j}]_{l\in[n],\,j\in[d]:=\{1,\ldots,d\}},$$

and $\vec{c}\in\mathbb{R}^n$ (with components c_j) of one building block (see Figure 1) with d inputs and n outputs. In order to formulate rules (4)–(6) below, we need to bound the weights. Thus, we choose $-1\le w_{l,j},c_j\le 1$. Given are m input vectors $\vec{x}_1,\ldots,\vec{x}_m$ with d nonnegative components each. We denote component j of $\vec{x}_k$ with $x_{k,j}\ge 0$. The weights have to be chosen such that the m output vectors $\vec{o}_1,\ldots,\vec{o}_m$ are closest to given target vectors $\vec{t}_1,\ldots,\vec{t}_m$ in the l^1 norm $\sum_{k=1}^m\sum_{j=1}^n|o_{k,j}-t_{k,j}|$. To this

[1] http://yann.lecun.com/exdb/mnist/

Algorithm 1 Iterative backpropagation-like learning with MILPs

procedure LEARN WEIGHTS(training_input_data, training_ground_truth_data)
  Randomly initialize all weights
  accuracy := 0, last_accuracy := -1, target_values := training_ground_truth_data
  while accuracy > last_accuracy **do**
    last_accuracy := accuracy
    Compute all neuron outputs $\vec{o}$ for training_input_data
    for i := number of output layer **back to** number 1 of first hidden layer **do**
      Update weights of (output or hidden) layer i with LP/MILP, see Section 2.1:
        Minimize l^1 norm of differences between output values of layer i and
        target_values. Input values of layer i are fixed, weights are variables.
      if i > 1 **then**
        Compute optimal input values $\vec{x}$ of this layer (which are output values $\vec{o}$ of the
        preceding layer) using a second LP/MILP, see Section 2.2:
          Minimize l^1 norm of differences between output values of layer i
          and target_values. Weights of the layer are now fixed.
          Input values are variables.
        target_values := computed optimal input values
    For updated weights and training_input_data, update inputs and outputs of all neurons
    Re-compute accuracy
  if accuracy < 1 **then**
    Update weights with those belonging to best accuracy that occurred in while-loop
    Finally optimize weights of the last layer, see LP in Section 2.3.

end, we express difference $o_{k,j} - t_{k,j}$ via two nonnegative variables $\delta^+_{k,j}, \delta^-_{k,j} \geq 0$:

$$o_{k,j} - t_{k,j} = \delta^+_{k,j} - \delta^-_{k,j}. \tag{1}$$

This leads to the problem

$$\text{minimize} \sum_{k=1}^{m} \sum_{j=1}^{n} (\delta^+_{k,j} + \delta^-_{k,j}) \tag{2}$$

under following restrictions (3), (4), (5), and (6) that deal with computing $o_{k,j}$. For each $k \in [m]$ and $j \in [n]$ we compute $o_{k,j} = \sigma\left(a_{k,j}\right) \geq 0$,

$$a_{k,j} := c_j + \sum_{i=1}^{d} w_{j,i} x_{k,i}, \tag{3}$$

where $\sigma(x)$ is the ReLU function. Let $\tilde{M} := \max\{x_{k,j} : k \in [m],\ j \in [d]\}$. Both $o_{k,j}$ and $|a_{k,j}|$ are bounded by $d\tilde{M} + 1$. In Section 2.2 we determine new inputs not necessarily bounded by $\tilde{M}$ but by $1.1 \cdot \tilde{M} + 0.1$. Thus, values of $o_{k,j}$ and $|a_{k,j}|$ are generally bounded by

$$M := d \cdot (1.1 \cdot \tilde{M} + 0.1) + 1.$$

To implement the piecewise definition of ReLU, we introduce binary variables $b_{k,j}$ that model, for input $\vec{x}_k$, whether a neuron j fires (value 1) or does not fire (value 0), i.e., if the input of ReLU exceeds zero (cf. [2], [4], [3]):

$$-M(1 - b_{k,j}) \leq a_{k,j} \leq M b_{k,j}. \tag{4}$$

If $b_{k,j} = 1$, output $o_{k,j}$, $0 \leq o_{k,j} \leq M$, of neuron j equals $a_{k,j}$ for input $\vec{x}_k$. Otherwise for $b_{k,j} = 0$, the output $o_{k,j}$ has to be set to zero:

$$-M(1 - b_{k,j}) \leq o_{k,j} - a_{k,j} \leq M(1 - b_{k,j}), \tag{5}$$

$$0 \leq o_{k,j} \leq M b_{k,j}. \tag{6}$$

The MILP can be divided into n independent MILPs that calculate $d + 1$ weights separately for each of the n neurons of the layer.

In theory, these MILPs can be replaced by 2^m LPs as follows. Let $j \in [n]$. For each $k \in [m]$, we can add constraints $a_{k,j} < 0$ or $a_{k,j} \geq 0$ to avoid binary variables and obtain LPs. Then, the weights are determined by a smallest objective value of all problems.

We really replace the MILP of the last layer by a single LP, which is potentially much faster than the MILP: To test the network, we use ground truth data consisting of one-hot vectors. If the digit to be recognized is j, $0 \leq j \leq 9$, then the jth component is one, all other

components are zero. Since the prediction is an output closest to one, we can replace ReLU with the identity function to obtain a linear problem, i.e. $o_{k,j} := a_{k,j}$ without constraints (4)–(6) such that $o_{k,j}$ may be negative. Instead of objective function (2) we deal with

$$\text{minimize} \sum_{k=1}^{m} \sum_{j=1}^{n} \delta_{k,j}, \tag{7}$$

where

$$\delta_{k,j} := \begin{cases} \delta_{k,j}^{+} & : \quad \text{ground truth } t_{k,j} = 0 \\ \delta_{k,j}^{+} + \delta_{k,j}^{-} & : \quad \text{ground truth } t_{k,j} = 1. \end{cases} \tag{8}$$

In the linearized version (7) of the error functional, we do not consider $\delta_{k,j}^{-}$ in the case $t_{k,j} = 0$ because only positive ReLU inputs $a_{k,j}$ contribute to the error. Non-positive inputs would be set to zero by applying the ReLU function and then match $t_{k,j} = 0$.

2.2 Proposing layer inputs

After optimizing the weights of a layer, its input data are slightly adjusted to further minimize the output error of that layer. This is described in what follows.

To adjust the input of a layer, basically the same MILP/LP as before can be used. Now weights $W \in \mathbb{R}^{n \times d}$ and $\vec{c} \in \mathbb{R}^n$ of one building block (see Figure 1) with d inputs and n outputs are given and are not variable. We need to find m input vectors $\vec{x}_1, \ldots, \vec{x}_m$ each with d components (which are now variables $x_{k,j} \geq 0$, $k \in [m]$, $j \in [d]$), so that for given weights, the problem (2) is solved under constraints (1), (3), (4), (5), and (6) for all but the last layer. For the last layer, problem (7) is solved under restrictions $o_{k,j} = a_{k,j}$, (1), (3), (8). Only small adjustments of inputs promise not to lead to major changes in the weights in subsequent steps. This is important since we do not want to forget information that has already been learned. Let $\tilde{x}_{k,j}$ be the input of the layer previous to this optimization step. Then we add bounds

$$\max\{0, 0.9 \cdot \tilde{x}_{k,j} - 0.1\} \leq x_{k,j} \leq 1.1 \cdot \tilde{x}_{k,j} + 0.1. \tag{9}$$

The bounds are helpful beyond that. Because without them, the runtime for determining subsequent weights increases significantly. Inputs can be calculated independently for each of the m training input vectors.

Instead of adjusting inputs to optimally match desired outputs of one single layer, an alternative approach would be to consider all subsequent layers with the goal of minimizing the distance to the ground truth. With weights held fixed, this is a linear problem similar to the tasks in [2, 4], etc. However, it turned out that considering more than one layer is not necessary due to the chosen iterative approach.

2.3 Post-processing of the weights of the last layer

So far, the objective functions have been built on the l^1 norm, which is needed in particular for weight calculation of hidden layers. But now, in a final step (see Algorithm 1), we adjust the weights of the last layer with an LP by minimizing $\sum_{k=1}^{m} \sum_{j=1}^{n} (\delta_{k,j} - s_{k,j})$ where variables $\delta_{k,j}$ are defined in (8), and slack variables $0 \leq s_{k,j} \leq 0.49$ are additionally constrained by $s_{k,j} \leq \delta_{k,j}$. Thus, we allow deviations up to 0.49 so that zeroes and ones of ground truth vectors are still separated.

3 Results and batch learning

Due to runtimes of MILPs, we did not apply all steps of Algorithm 1 to all 60,000 training images but only to small subsets (batches) of one hundred images. However, the LP of the post-processing step is able to handle the complete training set. The outcome of Algorithm 1 depends strongly on the initialization of weights. A good random initialization of weights $w_{j,i}, c_j \in [-1, 1]$ leads to the results shown in Figure 2 for the 784-8-8-8-10 network. While one can experimentally determine a suitable initialization, a larger issue is that accuracy is low on all 60,000 images after training on 100 images. Therefore, we experimented with iterative batch learning. Algorithm 1 was applied to an initial batch of images 1-100, and weights were updated accordingly. Then the algorithm was applied to images 101-200 on these updated weights, etc. We use a simple idea to better remember previously learned images: We do not only initialize weight variables for a warm start with values from the preceding batch training, but we also limit weight changes of consecutive batch learning steps. Starting with the training of the second batch, each weight $w := w_{j,i}$ or $w := c_j$ is additionally bounded depending on the corresponding computed weight $\tilde{w}$ of the same layer for

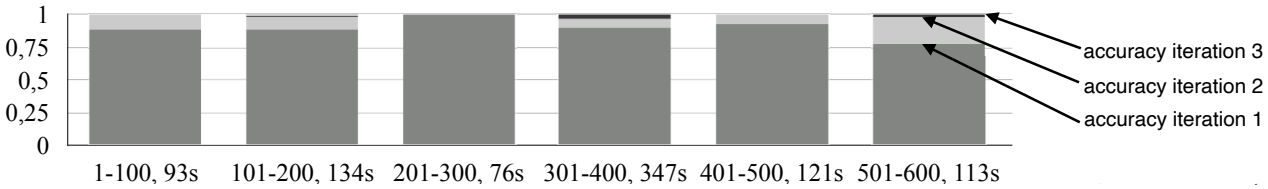


Figure 2: For training with 100 referenced MNIST images and randomly initialized weights, vertical bar segments show how (while-) iterations in Algorithm 1 increase the accuracy of the 784-8-8-8-10 network until a final accuracy of one is reached. The accuracy after the first iteration is shown at the bottom. Then the improvement of each subsequent iteration is added. The post-processing step was not required. Runtimes were measured with CPLEX 12.8.0 on a MacBook Pro with 16 GB RAM and an i5 processor (two cores).

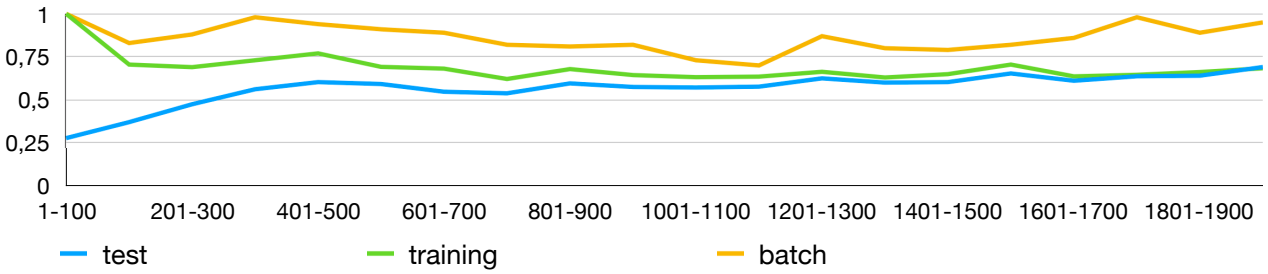


Figure 3: Iterative training of the 784-8-8-8-10 network with 20 batches of 100 consecutive images. Weights are bounded due to (10). The top curve shows the training accuracy with respect to each single batch. The middle curve represents the accuracy with respect to all training data seen so far. This consists of the current batch and all previous batches. The bottom curve visualizes the accuracy on the MNIST test dataset with 10,000 images. The runtime was 1,448 s.

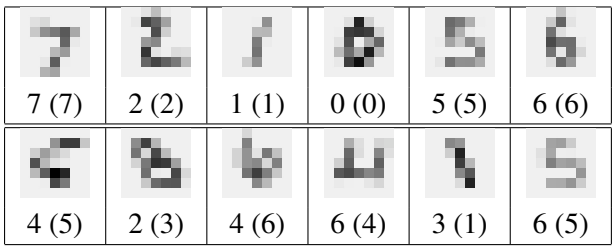


Figure 4: Reducing the resolution contributes to false detections. Ground truth is given in brackets. The best detection results (IoU values) were obtained for digit 1, worst for 5.

the preceding batch (factor 0.6 was determined experimentally):

$$\tilde{w} - 0.6 \cdot |\tilde{w}| - 0.01 \leq w \leq \tilde{w} + 0.6 \cdot |\tilde{w}| + 0.01. \quad (10)$$

This approach yielded a best case accuracy of 0.69 on the 10,000 test images (while Tensorflow/Keras reached a maximum of 73.3% after three epochs with the Adam optimizer, learning rate 0.001, batch size 100, all random seeds set to 4711), see Figure 3. Bound (10) also reduced processing times. We also trained with multiple epochs on shuffled data. To avoid overfitting, we added noise to ground truth vectors and tested a dropout strategy. However, all these methods did not significantly improve accuracy - unlike running the LP of the post-processing step on all 60,000 training images. To this end, we did not apply it for each batch but ran it after completing ten batches (i.e., training on 1000 images). It then achieved test accuracies of up to 75.84% within about a thousand seconds processor time. A majority vote of a committee of three networks trained on different sets of 1,000 images increased the accuracy to 79.19%.

For the convolutional network (randomly initialized with weights in $[0,1]$), we similarly trained weights iteratively on ten batches of 100 images (first 1000 images of training set) with rule (10) and then ran the post-processing step (Algorithm 1, Section 2.3) on all 60,000 training images in 3,933 seconds processor time to achieve an accuracy of 87.11%. Downsampling of image resolution was necessary to run MILPS in reasonable time, but reduces accuracy, see Figure 4. Using softmax activation on the last layer, the Adam optimizer (with parameters as before) achieved an accuracy of 93,51% on test data within 40 epochs when trained with full-resolution 28×28 images. However, after adding an average pooling layer to reduce reso-

lution consistent with MILP training to 7×7 pixels, only 89,61% accuracy is obtained in one minute (40 epochs). ReLU instead of softmax activation on the last layer implied worse accuracies up to 46%, thus MILP training performed better.

4 Conclusion

In this report, we have shown that it is possible to train networks iteratively based on MILPs. Accuracies as with the Adam Optimizer can be achieved. Thus, combinatorial optimization could be an alternative to gradient-based methods when they encounter difficulties. However, without further consideration, runtimes currently limit this approach to small training sets and simple networks. Future work may be concerned with the improvement of runtimes.

Acknowledgements

Many thanks to Christoph Dalitz for his valuable comments.

References

[1] D. Bienstock, G. Muñoz, and S. Pokutta, "Principled deep neural network training through linear programming," *arXiv*, vol. 1810.03218, pp. 1–26, 2018.

[2] M. Fischetti and J. Jo, "Deep neural networks and mixed integer linear optimization," *Constraints*, vol. 23, pp. 296–309, 2018.

[3] T. Serra, C. Tjandraatmadja, and S. Ramalingam, "Bounding and counting linear regions of deep neural networks," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 4558–4566, 2018.

[4] E. B. Khalil, A. Gupta, and B. Dilkina, "Combinatorial attacks on binarized neural networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[5] R. Anderson, J. Huchette, W. Ma, and et al., "Strong mixed-integer programming formulations for trained neural networks," *Mathematical Programming*, vol. 183, pp. 3–39, 2020.

[6] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[7] V. Dua, "A mixed-integer programming approach for optimal configuration of artificial neural networks," *Chemical Engineering Research and Design*, vol. 88, no. 1, pp. 55–60, 2010.

[8] D. Golovashkin and P. Aboyoun, "Minimizing global error in an artificial neural network," *U.S. Patent*, vol. US20150088795A1, pp. 1–8, 2015.

[9] Y. L. L. Cun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of IEEE*, vol. 85, no. 11, pp. 2278–2324, 1998.

[10] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.