Neural Network Safety
Formally Verifying Properties of Neural Networks Using Reluplex/Marabou

Rinkel, Leon
leon.rinkel@st.ovgu.de

Zeller, Josefine
josefine.zeller@st.ovgu.de

Abstract

Neural networks are becoming more and more popular and successful for solving all kinds of problems like natural language processing or image recognition. While there would be great benefit to it, it is still complicated to use them in critical systems like autonomous vehicles. Neural networks pose the risk of adversarial attacks, a way of fooling the network to misinterpret its input. Although networks are trained to defend adversarials and tested to have high accuracies, it is difficult to formally guarantee their correctness. We evaluate Reluplex/Marabou as a tool to verify properties of deep neural networks.

1 The Reluplex Algorithm

The reluplex algorithm aims to verify properties of deep neural networks. Given a linear conjunction of atoms and a network, the algorithm decides about the satisfiability [1]. The atoms set ranges for the input and expected bounds for the output variables. The network has to be a feed forward neural network with a piecewise linear activation function. Hence, the algorithm got the name because of the first implemented activation function, the rectifier function.

To decide whether a property is satisfiable, reluplex utilizes the well-known linear programming method simplex [7]. Simplex starts on a vertex of a polyhedron $P \in \mathbb{R}^n$ with

$$P = \{x \in \mathbb{R}^n | Ax \leq b, x \geq 0, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m\}$$

and moves along the edges to find the vertex where the objective function has the optimal value. Therefore a finite set of linear equalities is required. As there is no need for optimization, rather compute if a feasible solution exists, only phase one simplex is applied. Phase one identifies a starting vertex, if one exists, and phase two would optimize based on an objective function.

1.1 Transforming neural networks into linear programs

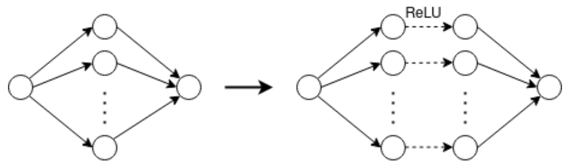
To apply simplex on neural networks, the network has to be described as a finite set of linear equalities. Therefore, considering the network layer by layer, each neuron $v_{n,i}$ in layer n is represented by a linear combination of the input nodes $v_{n-1,m}$ and their weights $w_{m,i}$.

$$\sum_{k=1}^{m} v_{n-1,k} \cdot w_{k,i} = v_{n,i}$$

The activation function is incorporated by splitting every hidden neuron into a pre and a post activation function node and creating a constraint for these nodes.

In case of the rectifier function follows $v_{post} = max(0, v_{pre})$.

Preprint. Under review.



To start the simplex method, an artifical vector $z \in \mathbb{R}^m_+$ is added to the system, creating an auxiliary problem

$$Ax + z = b \quad x, z \geq 0.$$

For this new system is $x = 0, z = b$ obviously a feasible solution. To verify if the original system has a feasible solution, the simplex algorithm can be applied with an objective function $min \sum_{i=1}^{m} z_i$. If simplex finds a solution, where $x = x^*, z = 0$, x^* is a basic feasible solution of the original problem an could be used as the starting vertex.

For solving a linear program, the problem is usually written as a so-called simplex tableau

$$\begin{array}{ccc|c} x_1 & ... & x_n & \\ \hline & A & & b \\ \hline \overline{c}_1 & \cdots & \overline{c}_n & \alpha \end{array}$$

with $x_1, ..., x_n$ as variables, $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^m$ representing the equality equations, $\overline{c}$ the vector of reduced costs and α the negative objective value.

1.2 Simplex Operations

Initially we have, as mentioned above, $z = b$ as basic variables and $x = 0$ as non basic variables. Because z is a artifical vector, the original problem has to have a basis only depending on variables of x. To minimize the objective function, basic variables can be replaced with non-basic variables.

A x_i, the pivot element, can enter the base and in exchange one z_j leaves. These pivoting steps are defined by the following rules.

Given a tuple $< \mathcal{B}, T, l, u, \alpha >$, with $\mathcal{X}$ as given set of variables, $\mathcal{B} \subseteq \mathcal{X}$ the set of basic variables, the lower bounds l and upper bounds u and α assigning a real value to each $x \in \mathcal{X}$.

To reduce the complexity of the following rules, two sets $slack^+(x_i)$ and $slack^-(x_i)$ are defined. Where x_i is a basic variable determined by the linear combination $x_i = \sum c_j x_j$ of non-basic variables x_j and their weights c_j.

$slack^+(x_i)$ is the set of non-basic variables x_j whose associated $c_j > 0$ and assignment is smaller than the upper bound, or $c_j < 0$ with $\alpha(x_j) > l(x_j)$. Analogue is $slack^-(x_i)$ the set of non-basic variables x_j with $c_j < 0 \wedge \alpha(x_j) < u(x_j)$ or $c_j > 0 \wedge \alpha(x_j) > l(x_j)$.

The Update rule is applied to make sure that non-basic variables satisfy their bounds.

$$\text{Update} \; \frac{x_j \notin \mathcal{B},\; \alpha(x_j) u(x_j),\; l(x_j) \leq \alpha(x_j) + \delta \leq u(x_j)}{\alpha := update(\alpha, x_j, \delta)}$$

To exchange a basic variable with a non-basic variable, two pivoting rules are defined.

$$\text{Pivot}_1 \; \frac{x_i \in \mathcal{B},\; \alpha(x_i) < l(x_i),\; x_j \in \text{slack}^+(x_i)}{T := pivot(T, i, j),\; \mathcal{B} := \mathcal{B} \cup \{x_j\} \backslash \{x_i\}}$$

$$\text{Pivot}_2 \; \frac{x_i \in \mathcal{B},\; \alpha(x_i) > u(x_i),\; x_j \in \text{slack}^-(x_i)}{T := pivot(T, i, j),\; \mathcal{B} := \mathcal{B} \cup \{x_j\} \backslash \{x_i\}}$$

Pivot$_1$ is applied if the assignment of the basic variable violates its lower bound and Pivot$_2$ if the upper bound is violated.

If a basic variable violates its bounds but the corresponding set of $slack$ variables is empty, no Pivoting rule can be applied and the problem is unsatisfiable.

$$\text{Failure} \; \frac{x_i \in \mathcal{B},\; (\alpha(x_i) < l(x_i) \wedge slack^+(x_i) = \emptyset) \vee (\alpha(x_i) > u(x_i) \wedge \text{slack}^-(x_i) = \emptyset)}{UNSAT}$$

When every basic variable satisfies its bounds, the computation is completed and the original problem is satisfiable.

$$\text{Success}\ \frac{\forall x_i \in \mathcal{X}, l(x_i) \leq \alpha(x_i) \leq u(x_i)}{SAT}$$

1.3 ReLU Constraints

Neural networks aim to approximate complex functions, that most of the time are of non-linear nature. In order to do so they need to have non-linear activation functions. This also renders the verification problem non-linear and prevents the previously described simplex algorithm from proving properties on those networks. This chapter focuses on what additions to the simplex algorithm have been made, so that it also works with deep neural networks using the ReLU activation function.

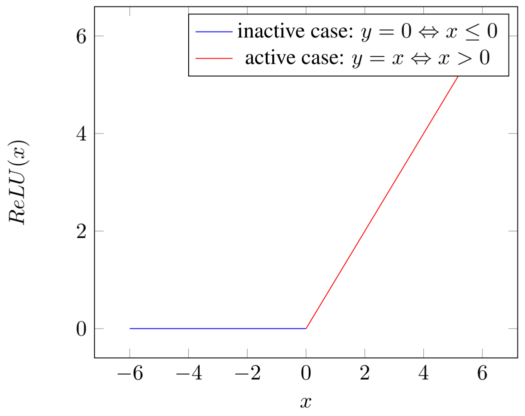


Figure 1: The ReLU activation function.

A naïve approach to handle the ReLU, or any other piecewise linear activation function, would be to combine the n linear terms of the activation function with disjunctions. That splits the problem into multiple sub-problems containing only linear equation sets. In the worst case this leads to an enumeration of the n^m different cases, m being the number of nodes, which is only practical for small networks.

For Reluplex however, the idea is to encode a node using the ReLU activation function as a pair of regular linear nodes, x_b being the backward-facing node taking the input and x_f being the forward-facing node producing the output, and then adding the following constraint connecting both nodes: $x_f = ReLU(x_b)$. In order to enforce these additional constraints, the $Success$ rule is replaced with the $ReluSuccess$ rule.

$$ReluSuccess[1]\ \frac{\forall x \in \mathcal{X}. l(x) \leq \alpha(x) \leq u(x), \forall \langle x^b, x^f \rangle \in R. \alpha(x^f) = max(0, \alpha(x^b))}{SAT}$$

The semantics of this representation is equivalent to the original but allows the solver to apply existing simplex rules to the backward- or forward-facing nodes independently. Updating one node of a ReLU pair might temporarily violate the ReLU constraint similar to violating the variable bounds. The $Update_b$, $Update_f$ and $PivotForRelu$ rules are introduced to correct a ReLU pair to satisfy the constraints in case of violation.

$$Update_b[1]\ \frac{x_i \notin \mathcal{B}, \langle x_i, x_j \rangle \in R, \alpha(x_j) \neq max(0, \alpha(x_i)), \alpha(x_j) \geq 0}{\alpha := update(\alpha, x_i, \alpha(x_j) - \alpha(x_i))}$$

$$Update_f[1]\ \frac{x_j \notin \mathcal{B}, \langle x_i, x_j \rangle \in R, \alpha(x_j) \neq max(0, \alpha(x_i))}{\alpha := update(\alpha, x_j, max(0, \alpha(x_i)) - \alpha(x_j))}$$

The $Update_b$ and $Update_f$ rules are for updating the backward- and forward-facing nodes of a ReLU pair, respectively. Due to the fact that ReLU activation is always greater than or equal to 0, the $Update_b$ rule may only be applied if the forward-facing node is within that range, otherwise it has to be corrected using the $Update_f$ rule first.

The update rules may only be applied if the node to be updated is non-basic. In case a ReLU constraint is violated and the corresponding variables are basic, the $PivotForRelu$ rule allows a basic variable of a ReLU pair to be pivoted with a different non-basic variable so that either $Update_b$ or $Update_f$ can be applied.

$$PivotForRelu[1] \quad \frac{x_i \in \mathcal{B}, \exists x_l. \langle x_i, x_l \rangle \in R \lor \langle x_l, x_i \rangle \in R, x_j \notin \mathcal{B}, T_{i,j} \neq 0}{T := pivot(T, i, j), \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}}$$

2 Specifying Properties for Neural Networks

2.1 Simple Net

When training neural networks, the training success is usually determined by a test data set. But that provides only knowledge of discrete points of continuous functions. Possible outliers between these points can cause unexpected behaviour if applied to a unknown data set.

To make sure that the approximated function does not behave differently between these points, Marabou provides the possibility to verify the function values of given intervals. Two demonstrate the application on an easy example, we trained the two dimensional function

$$f(x) = \frac{1}{x^2 + 1}.$$

Verifiable properties set bounds on the input and output variables. Marabou will then compute if a solution exists and either print 'sat' with a feasible solution or 'unsat'.

Our example has a global maximum at $x = 0, y = 1$ and $y = 0$ is the horizontal asymptote. Therefore, it could be interesting to verify, that the output always lays in the interval $(0, 1]$. To formulate valid properties, the proposition has to be transformed to check whether a solution exists that lays outside of these bounds.

For a function trained in the interval $[-5, 5]$ these properties could be:

$$\exists x, y : x \in [-5, 5] \implies y \in [1.1, \infty)$$
$$\exists x, y : x \in [-5, 5] \implies y \in (-\infty, -0.1]$$

If a solution exists that satisfies one of these formulas, our original assumption does not hold and the net has to be retrained.

2.2 Adversarial Robustness

Adversarial examples are input samples to neural networks or machine learning algorithms in general, that have been intentionally modified to fool the algorithm. For an image classifier for example, those samples often still look normal to the human eye, but contain small perturbations that cause a huge shift in predicted probabilities.

In this example we will (i) define adversarial robustness and (ii) use Reluplex to try to prove robustness for a simple MNIST classifier MLP and (iii) find adversarial examples as they are counterexamples to the robustness property.

There are multiple ways of defining adversarial robustness and even combinations of those, each with their own advantages and disadvantages. One key difference is the covered input range. Local robustness states that for an input x, which is close to a given input c, the networks classification $N(x)$, $N(c)$ is "similar" ($\forall x : \|x - c\| \leq \delta \implies N(x) = N(c)$). [2] [3] Global robustness instead covers an entire input domain D ($\forall x_1, x_2 \in D : \|x_1 - x_2\| \leq \delta \implies N(x_1) = N(x_2)$). [3] While both properties can be encoded as an input to Reluplex, the latter is significantly harder to prove. To encode the global robustness property, one would have to encode the network two times, one for $N(x_1)$ and another for $N(x_2)$, since x_1 and x_2 are no more constant but variable. [3] This doubles the number of trainable variables and input parameters. As we will see later on, local robustness is already a complex property to verify, we will therefore skip global robustness for now. Another difference is the distance metric used to determine how close two input points are. Given two samples

x_1 and x_2, the samples are δ-close to each other if $\|x_1 - x_2\| \leq \delta$. In order to encode this property as an input to Reluplex, the distance norm has to be composed of linear terms. That for example applies to L_1 and L_∞.

$$\|x_1 - x_2\|_1 = |x_{1,1} - x_{2,1}| + |x_{1,2} - x_{2,2}| + ... + |x_{1,n} - x_{2,n}|$$
$$\|x_1 - x_2\|_\infty = max(|x_{1,1} - x_{2,1}|, |x_{1,2} - x_{2,2}|, ..., |x_{1,n} - x_{2,n}|)$$

Figure 2: The L_1 and L_∞ norms.

When using the L_∞ norm, due to the max operator, only the highest pixel value difference is taken into account. So each pixel of the adversarial sample has to be in δ-range to the pixel value of the original image. Using this norm, with a reasonable parameter δ, we can prove robustness against adversarial examples like x_2, that have a small amount of noise applied to the entire image. With the L_1 norm, all pixel values are taken into account for determining an overall difference. Using this norm, we can also prove robustness against examples like x_1, that are mostly identical to the original image, but have few spots with high pixel value difference.

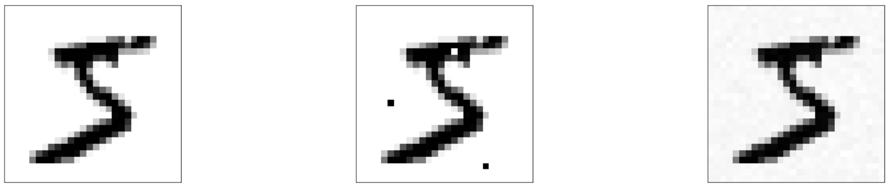

(a) Original image x_1 (b) Adversarial example x_2 (c) Adversarial example x_3

Figure 3: Different adversarial examples

There are also different definitions of what similar classification means. One might want to prove that an adversarial example is not misclassified, in other words, that the true label l_{true} still has the highest confidence C ($\forall l \in L \setminus \{l_{true}\} : C(x, l_{true}) > C(x, l)$) [3]. Another desired property is that there are no spikes in prediction confidence above a certain threshold ϵ ($\forall l \in L : |C(x_1, l) - C(x_2, l)| \leq \epsilon$) [3].

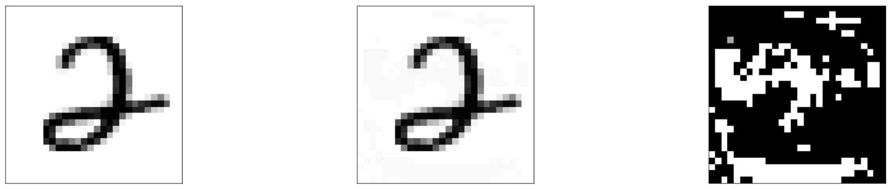
We use the following hybrid definition of adversarial robustness in our example: A model is (δ, ϵ)-locally-robust at a given point c, if there does not exist a δ-close point x, that is misclassified with a difference in confidence above a certain threshold ϵ. This property is then negated, so that the property holds if Reluplex resolves to $UNSAT$. For SAT, the solver yields a satisfying variable assignment, that can be interpreted as an adversarial example.

$$\neg(\forall x : \|x - c\|_\infty \leq \delta \implies \forall l \in L \setminus \{l_{true}\} : C(x, l_{true}) > C(x, l) + \epsilon)$$
$$\Leftrightarrow \quad \exists x : \|x - c\|_\infty \leq \delta \wedge \quad \exists l \in L \setminus \{l_{true}\} : C(x, l_{true}) \leq C(x, l) + \epsilon$$

Figure 4: Negating our definition of adversarial robustness.

The first part of the property ($\exists x : \|x - c\|_\infty \leq \delta$) is encoded as variable bounds of the input variables x_0 to x_n. While the lower bound L is set to the lower end of the δ-range ($L(x_i) = c_i - \delta$), the upper bound U is set to the upper end ($U(x_i) = c_i + \delta$). The second part of the equation is encoded as multiple sets of linear equations. Because Reluplex is not able to handle the non-linear softmax activation function, the property is evaluated on the networks output logits y_0 to y_9, that would be input to the softmax function. To encode the existential qualifier, the property is split into 9 sub-properties with the possible values for $l \in L \setminus \{l_{true}\}$, that are solved independently. If one of them resolves to SAT, the property as a whole is also satisfied.

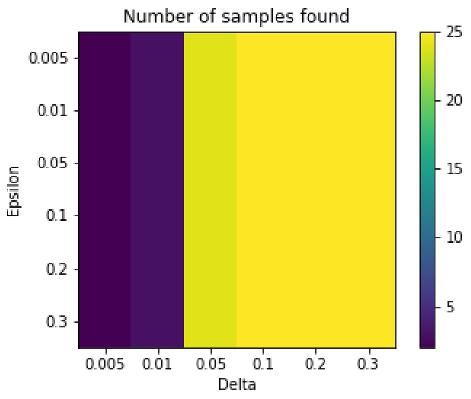
We trained a simple multilayer perceptron with 2 hidden ReLU layers, each with 16 nodes, to classify the MNIST dataset. After 3 epochs of training, the network achieved a validation accuracy of 93.6%. We then implemented the above steps in a Python script [10] and iterated over 500 samples of the MNIST dataset to find adversarial examples for our network. The time that the algorithm takes and the number of samples it will find, greatly depend on the network and the chosen parameters δ and ϵ. A network of the same network architecture, but with different weights due to random initialisation, will also have different samples and it might take significantly longer/shorter time to find those. In our particular case it took 45 minutes to find 104 adversarial examples. As expected, the network has 0% accuracy when validating it against these samples. Figure 5 shows one of the found samples.



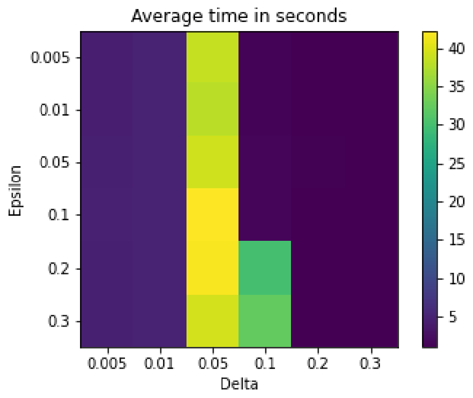
(a) Original image (b) Adversarial example (c) Difference (scaled)

Figure 5: Adversarial example found with the parameters $\delta = 0.01$, $\epsilon = 0.1$.

We measured the amount of adversarial examples found out of 25 given samples and the average time it takes to decide whether there exists an adversarial for a certain image, as a function of δ and ϵ. The measurements were made using the same MNIST MLP mentioned above. For each combination of parameter values, we use the same set of example images.



(a) Amount of adversarial examples found out of 25 given samples as a function of δ and ϵ.



(b) Average time in seconds it takes to decide whether there exists an adversarial example for a certain image as a function of δ and ϵ.

Figure 6

A δ value of 0.1 means, the allowed maximum pixel value difference is 10%. So if a pixel was completely white in the original image, it might be up to 10% black in the adversarial example. For higher values of δ, there may be more perturbation in the image. Figure 6a shows that there is a threshold of $\delta \geq 0.1$, that causes the algorithm to always find 25 out of 25 adversarial examples, due to high perturbation. On the other side, for values $\delta \leq 0.01$, it only found less than 5 samples. For both of these δ-ranges, Marabou solves in relatively short time (Figure 6b). Unfortunately, for reasonable parameter values - no visible perturbation ($\delta \approx 0.05$) and high shift in probability ($\epsilon \approx 0.3$) - it takes the longest time.

3 Performance evaluation

The worst-case number of pivoting steps is exponential for almost all known pivoting rules, but polynomial on the average and in most practical applications [7]. To evaluate the performance and scalability of marabou, we trained the non-linear two dimensional function $f(x) = sin(2x) + 1$ in the range $[-5, 5]$ and measured the runtime with different parameters.

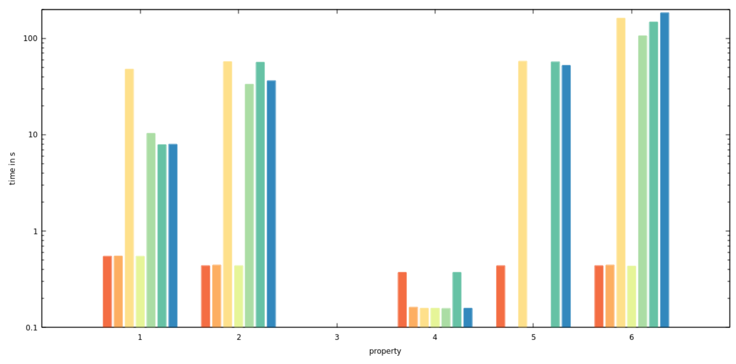
The property selection has a high impact on the runtime. Therefore, we evaluated the behavior of six properties, three that are expected to be satisfied and three unsatisfied. The first ones seem to be trivial, as either every point lays in the range or no point comes even close. The second ones evaluate the performance of properties in very small ranges and the last ones verify properties with only one bound on the output variable.

All measurements are the average of three runs on a Ryzen 7 pro 4750u CPU with eight cores, hyperthreading and 32 GB of RAM.

sat	unsat
$\exists x, y : x \in [-5, 5], y \in [0, 2]$	$\exists x, y : x \in [-5, 5], y \in [4, 6]$
$\exists x, y : x \in [0.6, 0.9], y \in [1.8, 2.0]$	$\exists x, y : x \in [0.6, 0.9], y \in [0.0, 0.2]$
$\exists x, y : x \in [-5, 5], y \in [1.9, \infty)$	$\exists x, y : x \in [-5, 5], y \in [-\infty, -0.1]$

3.1 Unpredictable Runtime

To show the impact of the weight initialisation and therefore the randomness of the number of required pivoting steps, we trained the same net seven times with random initial weights.



(a) Runtime with random weights

property	tf	onnx
1	0	0
2	0	0
3	7	5
4	0	0
5	3	3
6	0	0

(b) False classified

Figure 7

As expected shows Figure 7a a significant variation of the runtime across the seven nets and the different properties. The unsatisfied properties take slightly longer than the satisfied ones and smaller ranges seem to decrease the time to optimize the problem. Net one, two and four have an unusual fast runtime. All nets give the wrong output with property four and some with property 5. As marabou does not differentiate between error states and unsat properties, these nets probably raised an error. Converting the saved models to the onnx format improved the results slightly (Figure 7b).

3.2 Number of Neurons

The number of nodes determines the number of simplex and ReLU constraints. We trained eight nets with approximately 500 to 1500 hidden neurons.

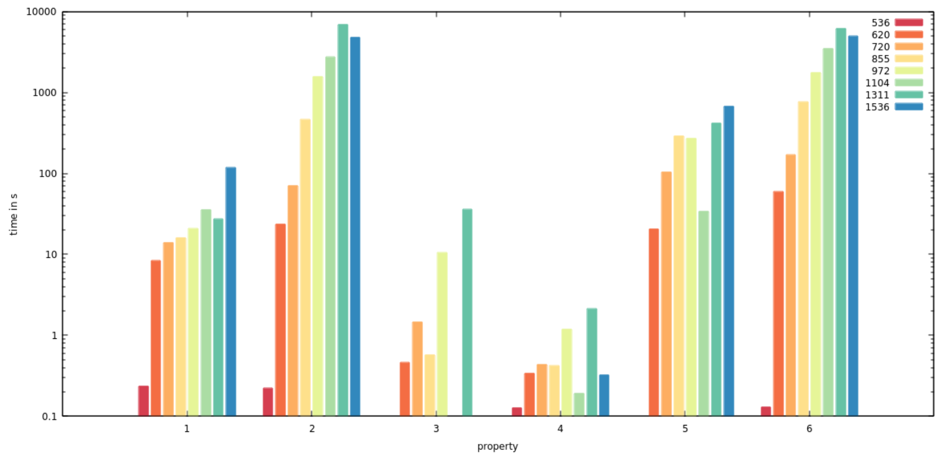


Figure 8: Runtime with increasing number of nodes

With increasing number of neurons the runtime rises exponentially. As seen in 4.1, smaller ranges reduce the problem size even though the dimension stays constant.

3.3 Parallel Execution

Marabou provides several parallelization techniques [9]. All are based on the Split-and-Conquer partitioning algorithm. Based on a strategy the algorithm divides the original problem into multiple sub-problems and tries to solve them in a fixed amount of time. If solving the sub-problem exceeds the specified time, the sub-problems is divided into multiple subsub-problems. The original problem is satisfied, if one of the sub-problems is satisfied. To influence the behavior, the user can set the number of initial splits and the number of workers.

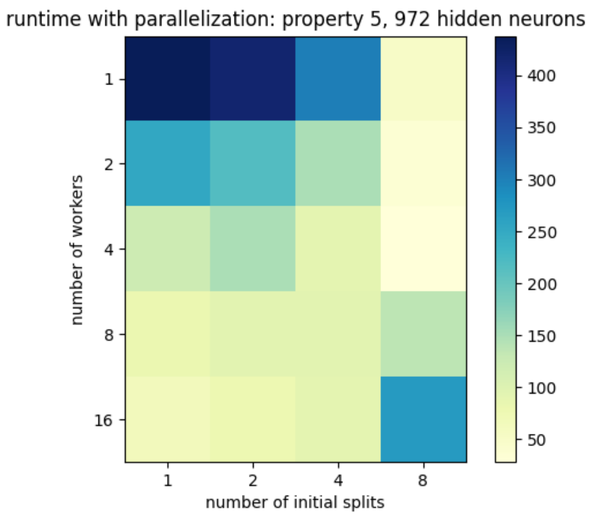


Figure 9: Runtime with parallelization in snc mode

Figure 9 shows the performance of propertie 5 with the net with 972 hidden nodes. Increasing the number of workers reduces the runtime on average. The specific problem performs the best with 4 workers and 8 initial splits. The runtime improved from 266s in dnc mode to 28s.

References

[1] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: "Reluplex: An efficient SMT solver for verifying deep neural networks", 2017. `https://arxiv.org/pdf/1702.01135.pdf`

[2] A. Pahlavan, D. Lee, J. Rose. "Defending the First-Order: Using Reluplex to Verify the Adversarial Robustness of Neural Networks Trained against Gradient White Box Attacks", 2018. `http://cs229.stanford.edu/proj2018/report/101.pdf`

[3] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. "Towards proving the adversarial robustness of deep neural networks", 2017. `https://arxiv.org/pdf/1709.02802.pdf`

[4] N. Carlini, G. Katz, C. Barrett, D. L. Dill. "Provably Minimally-Distorted Adversarial Examples", 2018. `https://arxiv.org/pdf/1709.10207.pdf`

[5] Marabou source code repository on GitHub. `https://github.com/NeuralNetworkVerification/Marabou`

[6] Katz, G., et al. "The marabou framework for verification and analysis of deep neural networks". In Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp.443–452. Springer, Cham (2019).

[7] Alexander Schrijver. "Theory of linear and integer programming". John Wiley & Sons, 1998.

[8] Vaŝek Chvátal. "Linear Programming". W. H. Freeman and Company, 1983.

[9] Wu, Haoze, et al. "Parallization Techniques for Verifying Neural Networks". 2020.

[10] Adversarial example generator script on GitHub. `https://github.com/leonrinkel/nns/blob/main/AdversarialGenerator.ipynb`