Increasing the efficiency of fuzzy logic and neural network algorithms

Thesis for the Degree of Doctor of Philosophy (PhD)

by Tibor Gábor Tajti

Supervisor: Dr. István Fazekas

University of Debrecen
Doctoral Council of Natural Sciences and Information Technology
Doctoral School of Informatics

Debrecen, 2020

Increasing the efficiency of fuzzy logic and neural network algorithms

Dissertation submitted in partial fulfilment of the requirements for the doctoral (PhD) degree in Informatics

Written by Tibor Gábor Tajti certified software engineer mathematician and computer science teacher

Prepared in the framework of the doctoral school of informatics of the University of Debrecen
(Theoretical foundation and applications of information technology and stochastic systems programme)

Dissertation advisor: Dr. István Fazekas

The official opponents of the dissertation:

	Dr.	....
	Dr.	....
	Dr.	....

The evaluation committee:

chairperson:	Dr.	....
members:	Dr.	....
	Dr.	....
	Dr.	....
	Dr.	....

The date of the dissertation defence: 202.. ....

Contents

1 Introduction and motivation **1**

2 Objectives and theses **2**

2.1 Objectives 2

2.2 Theses 4

3 Preliminaries **6**

3.1 Artificial intelligence and machine learning 6

3.2 Expression trees 6

3.3 Boolean logic, Short Circuit Evaluation in Boolean logic 7

3.4 Fuzzy logic, short-cut in fuzzy logic 8

3.5 Gödel fuzzy logic 9

3.6 Product logic 10

3.7 Łukasiewicz logic 12

3.8 Neural networks 13

3.9 Multilayer perceptron (MLP) 13

3.10 Convolutional neural networks (CNN) 14

3.11 Committee machine, voting functions 15

3.12 Ensemble methods in machine learning 17

3.13 Handling noise and avoid overfitting 19

3.14 MNIST (Modified National Institute of Standards and Technology database) 19

4 Fuzzy logic expression tree generation **21**

5 New fast evaluation algorithms for fuzzy logic formula trees **25**

5.1 Short circuit evaluations in Gödel type fuzzy logic 25

5.2 Algorithm with short-cuts on Product fuzzy logic 30

5.3 Fast evaluation algorithms for Łukasiewicz fuzzy logic 35

6 Simulation results for fast evaluation of fuzzy logic formula trees **41**

7 Improvements for neural network classifiers **58**

7.1 Fuzzification of neural network training data class membership crisp values 58

7.2 New committee machine voting functions 62

8 Performance evaluation of fuzzification and voter functions 65
8.1 Performance evaluation framework 65
8.2 Performance evaluation of training data class membership value fuzzification 69
8.3 Performance of voting functions 83

9 Conclusions and future work 92

10 Summary 93

11.Összefoglalás 95

12 Publications 97

13 References 99

14 Appendix 104

1 Introduction and motivation

In the first two decades of the 21st century, information has become a basic need, with a multitude of computers and the Internet able to store and make vast amounts of information easily accessible. The increasing performance of computers enabled the widespread use of artificial intelligence and machine learning technologies, these are coming into our daily lives, with image recognition, automatic translation, AI assistants, chatbots, autonomous cars, etc.

Fuzzy logic and artificial neural networks are two fields of the area of modern artificial intelligence, providing various algorithms with the ability of learning and providing decision support for us.

Fuzzy logic has been used in many applications such as machine control, knowledge-based systems, optimization problems, weather forecasting, risk assessment, medical diagnosis and treatment plans, etc. Some of its applications can need the processing of very large formulas.

Neural network algorithms are supervised machine learning algorithms, widely used in machine learning. Its major applications include classification, regression, pattern recognition, function approximation, intelligent control, learning from data.

Although the performance of the hardware is continuously increasing, it can also be possible to improve the efficiency of our algorithms and software applications using those algorithms. For a complex situation, the performance of our machine learning system can be dependent on the quality and performance of the sensors, the performance of the network, the performance of the hardware, the performance of the machine learning algorithms we are using, and the performance of our application as well.

Enhancements made in the machine learning algorithms can expand the range of problems for which we can run our algorithms efficiently on the presently available computers. Better efficiency also can help to make them usable on lower performance hardware and can make the application of machine learning more affordable, or can enable better accuracy if higher performance is available.

2 Objectives and theses

This section summarizes the author's major works and achievements in the development of fuzzy logic and neural network algorithms.

According to the tradition of the Ph.D. dissertation, we will use the already published results where appropriate using the editorial "we" for the description of the presented research.

In this part objectives and theses will be presented. Objectives are short descriptions of the research with mentions of the author's contribution. The theses list the new results which are the outcome of the performed research.

2.1 Objectives

O. I We conducted research and development on multiple problems. First, we worked on Boolean satisfiability problems, developing a problem generator. Then we started our research to develop fast evaluation algorithms for fuzzy logic expression trees. Our goal was to find techniques to make short-cuts in the recursive evaluation algorithm, in addition to the trivial short-cut possibilities, which are used by the interpreters or compilers of programming languages to evaluate Boolean logic expressions. These algorithms cut the nodes or sub-trees from the evaluation which do not affect the final result of the formula. We can call them also short-circuit, short-cut or lazy evaluation as well. A problem generator was used in this research as well.

- We conducted research on the development of problem generator of Boolean satisfiability problems, where our goal was to generate weakly not solvable Boolean SAT problems [1]. The author took a minor role in this work, however, this development inspired the following research using problem generators.

- We have examined methods to optimize the evaluation algorithm of the Gödel fuzzy logic large formula trees, with cut-off the nodes from the evaluation which are not needed to calculate the final result. For this purpose several pruning techniques have been developed, which can be used during the evaluation of a formula tree, having many nodes intact, the value of which is not needed to determine the final value of the formula tree [2]. The author's contribution to this research was adding several corrections to the pseudo-code, developing the efficient program code for it, developing a tree generator to generate a large number of Gödel type formulae in a formula tree format for measurement of pruning algorithm efficiency, developing a framework to produce figures from the results of the measurements, writing the Simulation results section, and contributions to other sections. This research has been a starting point of the following researches performed on fast evaluation algorithms of fuzzy logic expression trees, but this dissertation does not include the results published there [2].

- We have done research and development to achieve fast evaluations for large formula trees of Product logic formulas by cutting from the evaluation the nodes and their children, if the values of them do not have an effect on the final result of the formula [3]. The author's main contribution to this work was the development of the compact algorithm which was used for the research, the simple framework to run the simulations on multiple machines, running the experiments, and presenting the results with generated figures.
- We developed lazy evaluation algorithm variants (i.e. evaluation which cuts the nodes or sub-trees from the evaluation which do not have an effect on the final result of the formula) for efficient calculation of the results of large Łukasiewicz fuzzy logic formulae [4]. The author's contribution to the publication was the development of the new algorithms with several variants, two of them were included in the article, the enhanced framework to generate formula trees with different shapes, producing the figures with an enhanced figure generation framework based on Python Pandas and contribution to the proof of correction of the algorithms.
- We have done new research on creating more efficient algorithm variants on Gödel fuzzy logic formulas. In this research, we have developed a compact recursive algorithm for the Gödel fuzzy logic which is a specialized variant of the algorithm presented in [3, 4]. Despite the compact size, the new algorithm has better efficiency than the first algorithm [2]. The author's contribution to this work included the development of the new expression tree generation framework and the fast evaluation algorithms, and performing and presenting the measurements.

O. II We conducted research on Neural network and Convolutional Neural Network classification algorithm advanced techniques using three different neural network models for performance evaluation. These techniques might be used for other classification algorithms as well. These methods can be used separately or together as well.

- We have examined possible techniques that can be used to improve the performance of machine learning algorithms with respect to the possible quality issues occurring in the training data labels, the issue of having only crisp (binary) membership values even when the class membership is unclear. The objective was the correction of the training data where the given crisp class membership values can be inaccurate or misleading for some samples. We examined the usability of fuzzification of the crisp class membership values during training [5, 6]. The author's individual research.

- We have researched the possible use of new variants of committee machine simple voting functions, and also new meta voting function(s) which can use the single voting functions as their input [7]. The intuition behind using meta voting functions was that if the accuracy is usually better if we combine the results of multiple learners, it might be also useful to combine multiple techniques to combine the results of them. The author's individual work.

2.2 Theses

T. I We defined new algorithms for evaluation of fuzzy logic expression trees of Gödel type fuzzy logic, Product fuzzy logic [3] and Łukasiewicz fuzzy logic formulas [4](see subchapters 5.1, 5.2, 5.3). The algorithms work by limiting the evaluation of the actual node, by lower and upper limits, to set the interval in which we are interested in the exact value of the node. By this technique, a high ratio of the nodes in a large formula tree can be pruned from the evaluation, without affecting the result of the evaluation. This will lead to an optimized algorithm needing less performance to evaluate the formula tree. For the performance evaluation, a formula tree generator framework was developed. The formula tree generator framework is described in chapter 4. The proposed algorithms are presented and described in chapter 5. The effectiveness and efficiency of the new fast evaluation algorithms are shown in chapter 6.

T. II We defined new algorithms for ensemble learners to achieve improved performance. The used techniques include a meta voting function and voting function variants to use with meta voting functions, as well as the fuzzification of training data class membership values by individual or ensemble knowledge of single or multiple learners. The proposed fuzzification of the training data crisp class membership values and the proposed new voting functions can be used separately and together as well. For the evaluation of the proposed algorithms, a performance evaluation framework was developed (see subchapter 8.1).

T. II/a We defined a simple algorithm that can be used to fuzzify the training data binary class membership values. This method can possibly be used to correct the imprecise or incorrect training data output values during the training. The proposed modification can be used for individual learners and also as an ensemble method for multiple learners for better performance, for this purpose we defined the single and the ensemble variants of the algorithm as well (see subchapter 7.1) [6]. The performance evaluation of using corrected training data output values including the ensemble variant is shown in subchapter 8.2.

T. II/b We defined new voting function variants to be competitors of the well-known ones. We also defined meta voting functions which use the output of the well-known committee machine voting functions and also our new variants as their input. Our proposed new voting functions and meta voting functions can be chosen to replace the well-known voting functions (see subchapter 7.2) [7]. The performance evaluation of our new committee-machine voting functions is shown in subchapter 8.3.

3 Preliminaries

3.1 Artificial intelligence and machine learning

Artificial intelligence has developed into a very important field of informatics. The classical AI included search algorithms, game theory solutions, mathematical optimizations, evolutionary computation [8, 9]. In the beginning, the exploitation of machine power was done by developing algorithms created by human experts, when special software solutions were created to look for the solution to the defined problem. This is a deductive approach when one finds the solution, i.e. create the algorithm to find the specific solution, from the problem formulation. However, in many fields (meteorology, physics, engineering, biology, chemistry, biophysics, sociology, or medical sciences) there is a vast amount of data that is difficult for people to understand and interpret.

To eliminate the limitation of needing specific algorithms for specific problems, the generalization of the algorithms was the next step, which led to emerging new theories that are capable to do more or less general machine learning. This is an inductive approach, by the means that the goal is to develop algorithms that are able to work without knowing the rules of the problem set given to them. They process the data, and they learn to produce the answer and will find out the rules or build a model that will be able to work on unseen data, providing appropriate answers on them.

This new area of artificial intelligence includes supervised, unsupervised, and hybrid learning with several algorithms like Support Vector Machine, K-Nearest Neighbors, Decision Support Tree, Fuzzy Decision Tree, Decision Forest, Long short-term memory, Artificial Neural Networks, Convolutional Neural Networks, Deep Neural Networks, etc. [8, 10, 11, 12]. Some of these algorithms usually provide their useful knowledge in such a way that it is difficult to logically justify each decision (i.e. what can be the thoughts behind the decisions), while other algorithms may have their model in a human-readable structure, like a decision tree. Among the most widely used machine learning algorithms are the Artificial Neural Network or its Deep and Convolutional algorithm variants. They build their knowledge by developing the weights between the neurons.

3.2 Expression trees

In several fields of mathematics and other sciences various formulae are used to describe some rules of the world. The used operators are mostly binary, and usually, unary operators are also allowed. Mathematical, logical, etc. expressions are often displayed by tree graphs. The structure of the expression is easy to understand: The main operator is put to the root of the tree (it is the topmost node of the tree graph). Nodes with unary operators have exactly one child, while binary operators have two children. In some cases since associativity allows us to do so, some operators may have more than two children, e.g., one node with operator "+" could have four children, let us say a, b, c, and d, representing the formula $a + b + c + d$. To evaluate an expression, one should start from the leaves of the tree. The values given there are used to evaluate every sub-formula and finally, the whole, original formula [13].

3.3 Boolean logic, short circuit evaluation in Boolean logic

The classical binary (two-valued) logic was mathematically formalized by Boole at the end of the XIX century, hence it is also called Boolean logic. There are two truth-values used in binary logic which can be thought of as yes and no, true and false, written as 1 and 0, T and F, $\top$ and $\bot$. This mathematical logic is applied in various disciplines. Here we mention only some parts very briefly [14].

The syntax of Boolean logic is usually defined inductively. The induction description starts with the atomic formula: an infinite number of logical variables exist. Each counts as an atomic formula. The symbols $\top$ and $\bot$ are also atomic formulas.

The inductive step is based on logical operations. Usually, in binary logic, conjunction, disjunction, and negation are defined. The conjunction and disjunction operators have two operands, while the negation operator has one operand. The implication is also used very often. If A and B are two logical formulae, then their conjunction $(A \wedge B)$, disjunction $(A \vee B)$ and implication $(A \rightarrow B)$ are also logical formulae. We call the formulae A and B the main subformulae of the original formula. The negation A by logic formula $\neg A$ is also a logic formula. In this work, we use the above four operators.

All logical formulae can be made from atomic formulae with a finite number of inductive steps. Logical formulae can be represented by their expression trees.

If we have a logical formula and the truth-values assigned to the propositional variables that appear, the formula can be evaluated based on the semantic rules of binary logic:

- A conjunctive formula $(A \wedge B)$ is true iff both A and B are true.
- A disjunctive formula $(A \vee B)$ is true iff at least one of the formulae A and B is true.
- An implication formula $(A \rightarrow B)$ is true iff A is false or B is true.
- A negation formula $\neg A$ is true iff the formula A is false.

In all other cases, the result value of the formula is false. In binary logic only these two values exist, *true* and *false*. Although in some programming languages the binary truth-values have been extended with *null*. This value behaves in the operations so, that every operation where there is a *null* operand will result in *null*, except in case of short-circuit evaluation (see below).

According to the rules of binary logic defined above, it can be possible that the value of one of the two main sub-formulae is sufficient to know the result of the original formula. This leads to the so-called short-cut (or short-circuit) evaluation technique, where some vertices of the formula tree may not need to be visited; as their value has no effect on the final value of the formula. These short-cut techniques are used widely in programming languages, helping to compute faster, omitting the evaluation of the not needed subformulae [13, 15].

For *AND* operation, it is not necessary to check all the operands or subformulae if any of them are already known to be false.

Similar to the *AND* operation described above, the *OR* operation can be terminated early if we know that one condition is true, the value of the other conditions does not need to be checked.

Similar to these two short-cut possibilities which are used by the compilers or interpreters of programming languages, the implication operation also enables short-circuit evaluation. If there is a given $A \rightarrow B$ implication, and we already know that the value of operand A is false, then we do not need to evaluate operand B, the result of the operation will be true regardless of it. Similarly, if we already know that the value of operand B is true, then we do not need to evaluate operand A, the result of the operation will be true.

3.4 Fuzzy logic, short-cut in fuzzy logic

In fuzzy set theory, each element may belong to a set to a degree which can take values ranging from the $[0, 1]$ closed interval [16]. The idea behind fuzzy logic is that we allow partial belongings of the elements to the subsets of a universal set [17]. Fuzzy sets have ambiguous boundaries and gradual transitions between defined sets and this makes it to be appropriate to deal with the nature of uncertainty [18]. Each fuzzy set is represented by a membership function. Intuition, rank-ordering, and inductive reasoning can be, among many ways, to assign membership functions to fuzzy variables [19, 20].

The standard set of the degree of truth of fuzzy logic is the real unit interval $[0, 1]$, the natural order of which is $\leq$, from total falsehood (represented by 0) to complete truth (represented by 1) through the continuity of intermediate degrees of truth. The most basic assumption of (mainstream) mathematical fuzzy logic is that operators must be interpreted truth-functionally through a set of degrees of truth. We assume that these truth functions behave classically at the extremal values of 0 and 1. The very natural behavior of conjunction and disjunction can be achieved by using $x \wedge y = min\{x, y\}$ and $x \vee y = max\{x, y\}$ for each $x, y \in [0, 1]$.

Short-cut evaluation is possible in fuzzy logic, too. Since for extremal values the fuzzy logic operations will have the same results as in the case of binary logic, the short-cut possibilities described for binary logic are available also for the fuzzy logic. For intermittent values the different types of fuzzy logic will behave differently, so the short-cut possibilities must be found specifically.

The most well-known propositional fuzzy logics are:

- Monoidal t-norm-based fuzzy logic (MTL)
- Basic propositional fuzzy logic (BL)
- Łukasiewicz fuzzy logic
- Gödel fuzzy logic
- Product fuzzy logic
- Fuzzy logic with evaluated syntax (sometimes also called Pavelka's logic)

In the following, we introduce three of the main fuzzy logic types in more detail, which are based on the basic t-norm.

3.5 Gödel fuzzy logic

This system was introduced by Kurt Gödel in 1932 [21]. Possible truth values are real numbers from the closed unit interval $[0, 1]$, i.e. $0 \leq x \leq 1$ real numbers. There are four main connectives defined for Gödel's system, negation, disjunction, conjunction, and implication, denoted by the symbols $\neg$, $\wedge$, $\vee$, and $\rightarrow$. Their syntax is the same as in Boolean logic and their semantics are defined as follows [22, 23, 24]:

$$|\neg A| = \begin{cases} 1, & \text{if} \quad |\mathrm{A}| = 0 \quad (1.1a) \\ 0, & \text{otherwise} \quad (1.1b) \end{cases}$$

$$|A \to B| = \begin{cases} 1, & \text{if} \quad |A| \leq |B| \quad (1.2a) \\ |\mathrm{B}|, & \text{otherwise} \quad (1.2b) \end{cases}$$

$$|A \wedge B| = min(|A|, |B|) \tag{1.3}$$

$$|A \vee B| = max(|A|, |B|) \tag{1.4}$$

The system is infinitely many valued and satisfies the axioms of intuitive logic with an additional law, namely the law of the chain: The formula

$$((A \to B) \vee (B \to A))$$

has a value of 1, regardless of the subformulae A and B.

We note that the above conjunction and disjunction operators can be referred as AND and OR and also as MIN and MAX. Expressions and thus expression trees in Gödel logic are very similar to that of Boolean logic. The difference is that the values in Gödel logic given on the leaves (i.e. the "real" values of the variables) are not limited to the logical values {0,1}, but real numbers between 0 and 1 (in the real sense) can be used. In subchapter 5.1 we present our fast evaluation strategy for Gödel fuzzy logic formulas.

3.6 Product logic

One of the well-known fuzzy logic systems is Product logic. This has been described specifically in a mathematical formulation in [25]. This logic interprets the conjunction by multiplication for the $[0, 1]$ closed interval [26, 27, 28]. If the values are limited to the traditional $\{0, 1\}$ binary set, the product is actually the same as the usual Boolean conjunction [3]. This logic is considered a fuzzy logic based on one basic t-norm since all continuous t-norms are locally isomorphic to the Product t-norm.

In product logic, the truth-values are the real numbers of the closed interval $[0, 1]$.

The syntax of Product logic is the same as the one of Boolean logic: negation is unary, implication, conjunction, and disjunction are binary operators. Formula trees can also be defined and used in a similar way.

The semantics of product logic is defined as follows. In general, variables and constants can have any value from the real interval $[0, 1]$, including the two classical values. The truth values of formulas with operators can be calculated from the value of their main subformulae [24, 25, 3]. We note that $max(A, B)$ is defined as a disjunction operator in [25]. Since it is the same as the disjunction operator used in Gödel type logic, we are interested in using the probabilistic sum as a disjunction operator, as presented as the dual t-conorm of the fundamental product t-norm $(A + B - AB)$ in [29].

$$|\neg A| = 1 - |A| \tag{2.1}$$

$$|A \to B| = \begin{cases} 1, & \text{if} \quad |A| \leq |B| \quad (2.2a) \\ |\mathrm{B}|/|\mathrm{A}|, & \text{otherwise} \quad (2.2b) \end{cases}$$

$$|A \wedge B| = |A||B| \tag{2.3}$$

$$|A \vee B| = |A| + |B| - |A||B| \tag{2.4}$$

[3]

Both conjunction and disjunction are associative in this logic, therefore, to make our work more efficient, we can allow more than two children of conjunction and disjunction nodes in the expression trees, similar to the case of Boolean logic.

In the product logic system, the conjunction is the product of the values of the arguments, and as defined above, we use the "probabilistic sum" as the disjunction. This type of conjunction and disjunction look more like operations of a probabilistic system [30]. Assuming that the values A and B are independent we get the result of their common occurrence.

$$P(A \; and \; B) = P(A)P(B)$$

$$\begin{aligned} P(A \; or \; B) &= 1 - P(\neg A \; and \; \neg B) = 1 - P(\neg A)P(\neg B) \\ &= 1 - (1 - P(A))(1 - P(B)) = P(A) + P(B) - P(A)P(B) \end{aligned}$$

The value of A→B is the maximal probability of B if A is true.

In a given (fixed) evaluation, the formula and its leaves are fixed, the leaves have labels from the closed interval $[0, 1]$. Then, the task is to compute the (truth) value of the whole formula. This can be done with a bottom-up strategy. However, as in the case of Boolean logic, we may not need to compute the value of each subformula to know the final result. In subchapter 5.2 we will show pruning techniques that can be used to quickly evaluate Product logic formulas [3].

3.7 Łukasiewicz logic

Łukasiewicz also applied the idea that takes into account intermediate truth values rather than a set of two elements for classical truth values $\{0, 1\}$. He invented three- and four-valued systems first, what he later extended to arbitrarily many ($n \geq 2$) truth values [31, 22]. As the various diverse logics, the infinitely many truth-valued Łukasiewicz logics are among the most attractive candidates for fuzzy logic [32]. In this system, all real numbers in the closed interval $[0, 1]$ can be truth values [33]. The language has two primitive logical connectives, i.e., $\rightarrow$, $\neg$, where $\rightarrow$ is the Łukasiewicz implication and $\neg$ is the negation operation. Based on those two operations the other connectives, the Łukasiewicz type conjunction ($\wedge$) and disjunction ($\vee$) were also defined. Here we use these four connectives and describe the system based on them. The syntax of this logic is exactly the same as the syntax of Boolean logic: negation ($\neg$) is unary, implication ($\rightarrow$), conjunction ($\wedge$), and disjunction ($\vee$) are binary connectives. Formula trees can also be defined and used similarly. The semantics of Łukasiewicz logic is defined in the following way. Generally, the variables and the constants may have any values from the closed real interval $[0, 1]$ including the classical two values. The truth-values of formulae with connectives can be computed from the value of their main subformulae [31, 32].

$$|\neg A| = 1 - |A| \tag{3.1}$$

$$|A \rightarrow B| = min(1 - |A| + |B|, 1) \tag{3.2}$$

$$|A \wedge B| = max(|A| + |B| - 1, 0) \tag{3.3}$$

$$|A \vee B| = min(|A| + |B|, 1) \tag{3.4}$$

[4]

We note that the above conjunction and disjunction operators are considered as the strong conjunction and disjunction operators. The weak variants of them are the same as in the Gödel type fuzzy logic, hence we do not perform the experiment with them in this research.

Łukasiewicz logic with similar semantics can also be used to have a finite number of truth values. In these systems, denoted by $L(k)$ for each integer $k > 2$, the values are: $0 = \frac{0}{k-1}, \frac{1}{k-1}, \ldots, \frac{k-1}{k-1} = 1$. In the special case $k = 2$, one gets back the classical Boolean connectives working on the classical truth values. The computation of the final value of a formula can be done with a bottom-up evaluation. As in the case of Boolean logic and the other fuzzy logic types described above, we do not need always to evaluate each sub-formulae to get the final result. In the subchapter 5.3 we present our fast evaluation strategy that can omit from the evaluation the nodes or sub-trees that do not affect the final result of the formula [4].

3.8 Neural networks

Neural network algorithms are supervised machine learning algorithms, widely used in machine learning. Their major applications include classification, regression, pattern recognition, function approximation, intelligent control, learning from data. The neural network is basically a set of interconnected artificial neurons and the appropriate algorithms working on them [10]. Simple and widely used neural network architecture is the Multi-Layer Perceptron (MLP) model.

3.9 Multilayer perceptron (MLP)

The basic architecture of an applicable neural network (MLP) is presented in Figure 1.

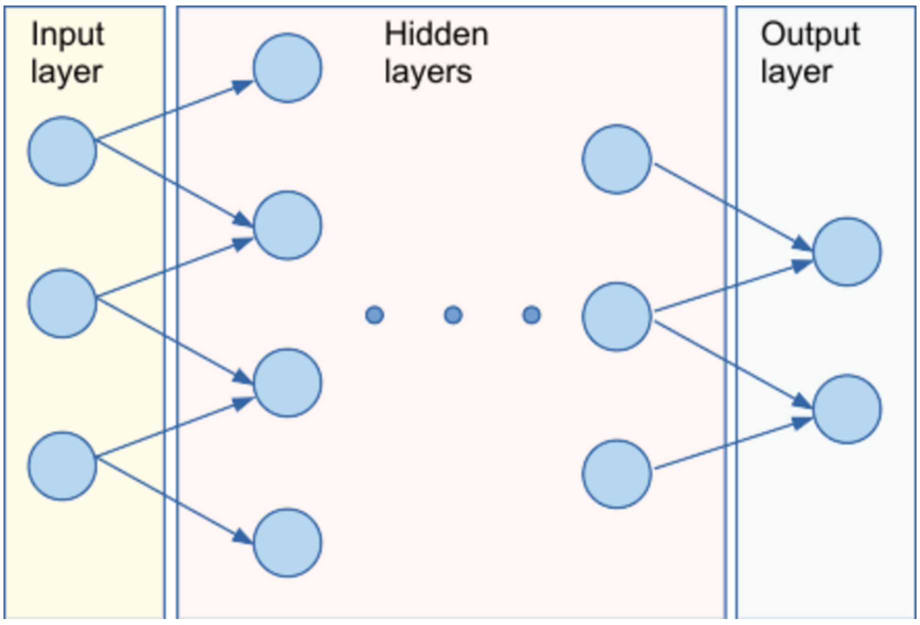


Figure 1: Simple multi-layer neural network architecture. (Drawn by the author)

1. Input: The sequence $x(n) = (x_1(n), ..., x_m(n))$. This is a vector of m elements, which can be considered the input signal, given at the nth time. These values are known to us, available, or observable and measurable.

2. Weights: They make the connection between successive layers. They are a

$$A_{m,n} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}$$

matrix of values where $\mathrm{w_{ji}}$ is the weight connecting the ith neuron from the previous layer to the jth neuron of the next layer, w_{ji} are used for its nth time weights. These values, at least the target values, are not known. It is our (the algorithm's) job to determine them. In each step, we approach the target value by approximation.

3. Neurons: Neurons of all other layers except the input layer get their values from the previous layers, so each layer but the output one will be the input to the next layer. The weighted sum of the nth time input data is formed according to the following formula, where i is the index of the source neurons 1 to m and j is the index of the target neuron:

$$y_j(n) = \varphi(b_j(n) + \sum_{i=1}^{m}(w_{ji}(n) \cdot x_i(n)))$$

The distortion (bias) was shown in the formula, although in real applications it is often replaced by a neuron with a constant 1 value, so its weight corresponds to the bias. Its nth approximation is formally denoted by $b_j(n)$. φ is the activation function, which converts the result of the summation to the appropriate interval [10].

In the MLP model successive layers are usually fully connected.

3.10 Convolutional neural networks (CNN)

A variation of the multi-layer perceptron model is the convolutional neural network. LeNet was one of the very first convolutional neural networks creating an area of deep learning. Yann LeCun's pioneering work has been named LeNet-5, after many successful iterations [34]. Convolutional networks have shown to be very effective e.g. in image classification [10, 35], natural language processing [36] and time series forecasting [37]. CNNs have a convolution operator, hence the name convolutional network. This convolution operator does feature extraction, e.g. when learning to classify a 2D image, smaller (e.g. 3x3 or 5x5 pixels) parts of the image will be processed as a sliding window over the whole image, so the network learns such smaller-scale features of the images. Neural network and Convolutional neural network algorithms are among the best performing machine learning algorithms. However, the performance of the algorithms may vary between multiple runs because of the stochastic nature of these algorithms. This stochastic behavior can result in worse-than-average accuracy for a single run, and in many cases it is difficult to decide whether we should repeat the learning, giving a chance to have a better result. Among the useful techniques to solve this problem, we can use the committee machine and the ensemble methods, which in many cases give better than average or even better than the best individual result [38].

Just as in human society combining the knowledge of experts can be very useful in machine learning as well. When several learner algorithms learn the same problem or parts of the problem, their knowledge can be combined in numerous ways [39, 11]. This can be used both for getting satisfactory results from weak learners and for reaching top performance when using strong learners. In the following two subchapters 3.11 and 3.12 some committee machine and ensemble methods are introduced.

3.11 Committee machine, voting functions

Committee machine algorithms use multiple instances of neural networks or other machine learning algorithms to make predictions and combine their results [40, 41]. This can work with multiple instances of the same algorithm (e.g. [35]) or different algorithms or models (e.g. [42]) as well. Several simple committee machine variants are used efficiently with the committee working on the same problem and combining their results with voting functions [43]. The most prominent ones are described in the following.

For each voting function, let o_i be the actual output vector of class membership values predicted by learner i for the actual sample given as input.

- **fuzzy average voting (V1 in chapter 8.3)**:

  Averaging is one of the most simple linear combiner voting schemes having the $1/N$ weight for the outputs of each learner [44].

  Calculate the average of the individual predictions:

$$o[j] = \frac{1}{N}\sum_{i=1}^{N} o_i[j]$$

for each j output class, where N is the number of learners, $o_i[j]$ is the jth element (class membership value) in the output vector of the prediction.

Then find for each sample the class with the highest class membership value as the chosen class for the given sample:

$$l = argmax(o)$$

- **plurality voting [45] (sometimes called majority voting) (V5 in chapter 8.3):**

  Find for each learner i the class with the highest membership value from the prediction o_i. If it is at index h_i ($h_i = argmax(o_i)$), then let

  $$c_i[j] = \begin{cases} 1, & if \quad j = h_i \\ 0, & \text{otherwise} \end{cases}$$

  for all j classes. Then calculate the sum

  $$s[j] = \sum_{i=1}^{N} c_i[j]$$

  for each j class, where N is the number of learners. The winner of the voting for the sample is a class with maximum value:

  $$l = argmax(s)$$

  We note, that sometimes this method is called majority voting, although majority voting means choosing the winner only if more than 50% of the learners have voted on it. When using majority voting it is recommended to use an odd number of voters.

- **Borda voting (Borda, 1784.) [46] (V6 in chapter 8.3):**

  For each individual learner i, calculate the index $s_i[j]$ in order of the membership values from the prediction o_i. Let $s_i[j]$ be n if $o_i[j]$ has the nth smallest value, for each j class for each i learner. Eventually $s_i = argsort(o_i)$. Then calculate the sum

  $$s[j] = \sum_{i=1}^{N} s_i[j]$$

  for each j class, where N is the number of learners used for the prediction. The winner of the voting is a class with maximum value:

  $$l = argmax(s)$$

- **Nash (product) voting [46](we will replace it with geometric mean in subchapter 7.2, (V7) in chapter 8.3):**

  For each j class evaluate the product of the predictions of all of the i individual learners:

  $$o[j] = \prod_{i=1}^{N} o_i[j]$$

  where N is the number of learners Then find for each sample the class with the highest membership value:

  $$l = argmax(o)$$

The plurality voting and the Borda voting are suitable for classification only, while the fuzzy voting and product voting can be used efficiently for regression as well.

These voting functions can be applied simply on the predictions of the individual learners that have learned either sequentially or in parallel.

Note that more voting functions are available, e.g. minimum, maximum, median voting [47]. We use the most well-known voting functions: fuzzy average, weighted fuzzy average, plurality, Borda, and product voting in our research.

3.12 Ensemble methods in machine learning

Another approach to combine the knowledge of several learners effectively is to use ensemble methods [48]. Ensemble methods also can be used in machine learning to learn a target function by training multiple individual learners and combining their results. The most popular ensemble methods are bagging, boosting, stacking, and random forest.

The ensemble methods can not only apply a voting function on the predictions of the individual learners combining their knowledge (see subchapter 3.11), but also they can control the learning process to reach a collaboration between the individual learners.

Ensemble methods have been very successful in setting record performance on challenging data sets [12]. We give a short introduction to some well-known ensemble methods below.

3.12.1 Bagging

The Bagging (bootstrap aggregating) method works by executing individual learners on different parts of the training data sets to achieve a diverse ensemble of learners. This algorithm works by selecting sample data set from training data, run it for a given number of iterations, then combine the learned classifiers. The bagging algorithm generates a different data set from the training data for each member of the ensemble. The predictions are combined either by uniform averaging or voting over class labels. The samples the individual learners are working on are called bootstrap; the name of the algorithm comes from chosen letters of Bootstrap AGGregatING. This algorithm makes a very good improvement even on weak learners' predictions. For the voting algorithm or formula, bagging uses both the averaging and the majority (or plurality) voting, the first for regression, the latter for classification [49].

3.12.2 Boosting

Boosting is similar to Bagging in that it also works by generating a diverse set of learners. It is another efficient algorithm which ensembles weak learners to achieve a strong learner. One of the best boosting algorithms is Adaboost, which is an adaptive boost algorithm. This algorithm iteratively creates weak learners and updates the weights of training data to combine the predictions of the weak learners by averaging to produce a strong learner. A boosting algorithm finds such a combination of weak learners which can produce much better accuracy than the individual participants. To achieve this it chooses sample data subsets from the training data strategically to get the most effective training data for each individual classifier. In the beginning, each data element has equal probability to be randomly chosen to be part of the training data for a learner. During training, the weights of the misclassified data will be increased to have a higher probability to get into the data set for the individual learners. Adaboost is a well-known and very efficient boosting algorithm [50].

3.12.3 Stacking

Wolpert's stacked generalization (or stacking) algorithm is a scheme to minimize the generalization error rate. It creates a set of learners in the first stage. These will be trained and used with the bootstrap ensemble method. They produce predictions which will feed the second stage learners, which can be considered as meta-learners. Their job is to learn how effective the first stage learning was. They can learn if the first stage learners have not learned particular pieces of the training data or training region and correct them when using for predictions. They also learn which first stage learners give correct output for the given input [51]. Unlike bagging and boosting, stacking may be (and often is) used to combine models of different types.

3.12.4 Random forest

A random forest [52] is an ensemble method working with (decision) tree learners of classification or regression trees. It uses the general bagging ensemble method on tree learners with the difference that the bagging usually can be used with various learners, in the case of the random forest, it will be used only with tree learners. Its individual learners usually can be considered very weak learners, its specialty is using a large number of individual tree learners [11].

3.13 Handling noise and avoid overfitting

Noise plays a significant role in teaching neural networks. On the one hand, adding noise can help generalization, i.e. avoid overfitting. Adding random distortion or data augmentation to training data are useful techniques widely used in machine learning. On the other hand, noise can make inaccuracies and can lead to misleading patterns or mislabeled training instances. The effect of inaccurate class labels in training samples has been shown by research to degrade the performance of even the best algorithms over a wide range of classification problems [53, 54]. It has also been observed that noisy labels are generally more harmful than noisy input patterns [55].

3.14 MNIST (Modified National Institute of Standards and Technology database)

Public domain databases help researchers with providing training and test samples for teaching and testing the machine learning algorithms. These include data sets from a variety of areas. Data sets containing labeled images are among the most well-known ones.

One of the most widely used datasets is the Modified National Institute of Standards and Technology database (MNIST) [56], which contains 60,000 handwritten numbers in the training set and 10,000 handwritten numbers in the test set. Different classifiers, like K-Nearest Neighbors, SVMs, Neural Nets, Convolutional Neural Nets, on this database have shown fail rate down to below 1%. State-of-the-art architecture as of the time writing this dissertation is the squeeze-and-excitation network [57, 58].

The MNIST database has samples that can be easily classified and hard (or impossible) ones as well.

We show examples from the easily recognizable samples below.

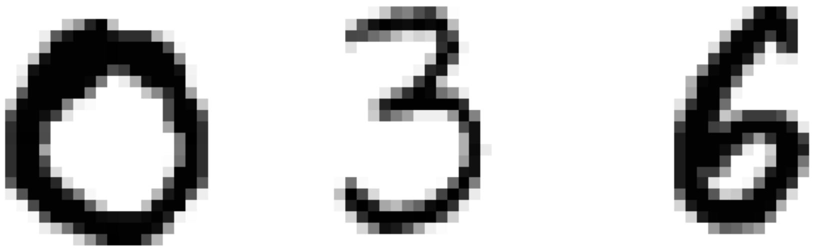


Figure 2: Easily recognizable images from the MNIST database test samples.

Figure 2 shows three chosen images from the test dataset of MNIST, that are easy to classify. In the case of such images, the binary class membership values of the classification are reasonable.

We also provide examples of hard-to-recognize images from the test samples.

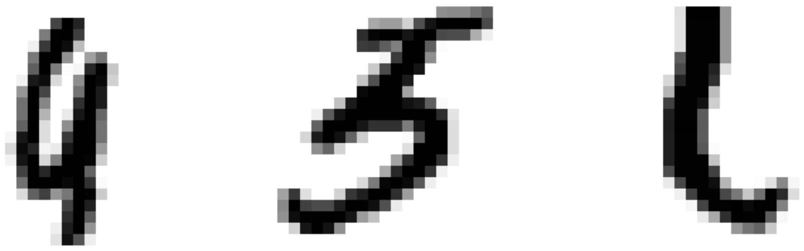


Figure 3: Hard-to-recognize images in the MNIST test dataset

Figure 3 presents three of the test images from the MNIST database that are among the ones which are not easily recognizable. Similar quality images are present in the training dataset as well. This quality issue is one of the reasons that motivate our research.

4 Fuzzy logic expression tree generation

We have developed a framework for our research to produce a large number of formula trees, on which the evaluation algorithms can be executed. The framework has been created for flexible purposes, allowing to set parameters for the distribution of values in the $[0, 1]$ interval, the shape of the tree, the allowed maximal number of child nodes for the node types, and the number of nodes [3, 59].

Since the various fuzzy logic types are three or more valued, many-valued, infinitely many-valued distributions, the ability to generate formula trees for such value distributions has been added to our framework. It can even generate two-valued formula trees for experimental purposes. Also, it can be used with random values generated from the $[0, 1]$ closed interval using the available accuracy provided by the machine and the programming environment.

The parameter setting to determine the shape of the formula tree has been added to be able to set a simulation environment to various needs since real-life cases show that the shape of formula trees can be very diverse. In some cases, we can find problems that can be described by balanced or roughly balanced trees, similar to a decision tree, which shouldn't have great depth for faster decisions. In other cases, the tree can be narrow and deep, e.g. in a communication network, or a wireless sensor network. The formula shapes that can be used by our generator:

- max 10 nodes at each tree level;
- max constant $\sqrt{N}$ nodes at each tree level, where N is the required number of nodes of the tree, given as parameter;
- max h nodes at the tree level of depth h ($h = 1$ for the root);
- max $10h$ nodes at the tree level of depth h;
- max h^2 nodes at tree level h;
- no limit is used (i.e., the natural limit 2^h nodes at depth level h for a binary tree);
- balanced tree, that is, the depth level of the leaves may differ by no more than 1.

The last two tree shapes are the most condensed ones, their depth can be thought of as minimal. These are extremal classes in our experiment. These two shapes are very similar since when generating the tree with the limit of 2^h nodes at depth level h, it will be naturally very close to the balanced tree. We also have a parameter for the maximal number of child nodes for each node type, which is provided for the case when operators can have more than

2 operands. In this research, we have not used operators with more than 2 operands.

The usable node types also can be set for the formula tree generation, for experimental or analysis purposes.

All the parameters described above can be used to determine the properties of the generated tree, which can be done for different tree sizes. Since our pruning algorithms have an overhead due to the extra computations and conditions, their advantage comes when we are using them for large expression trees. The node count parameter is used to determine the size of the tree. For our experiments, we have used the tree generator with tree sizes from 10 to 100000, effectively.

Now we will show the minimum and the maximum depth of the formula trees generated with the limits determining the tree shape, as described above.

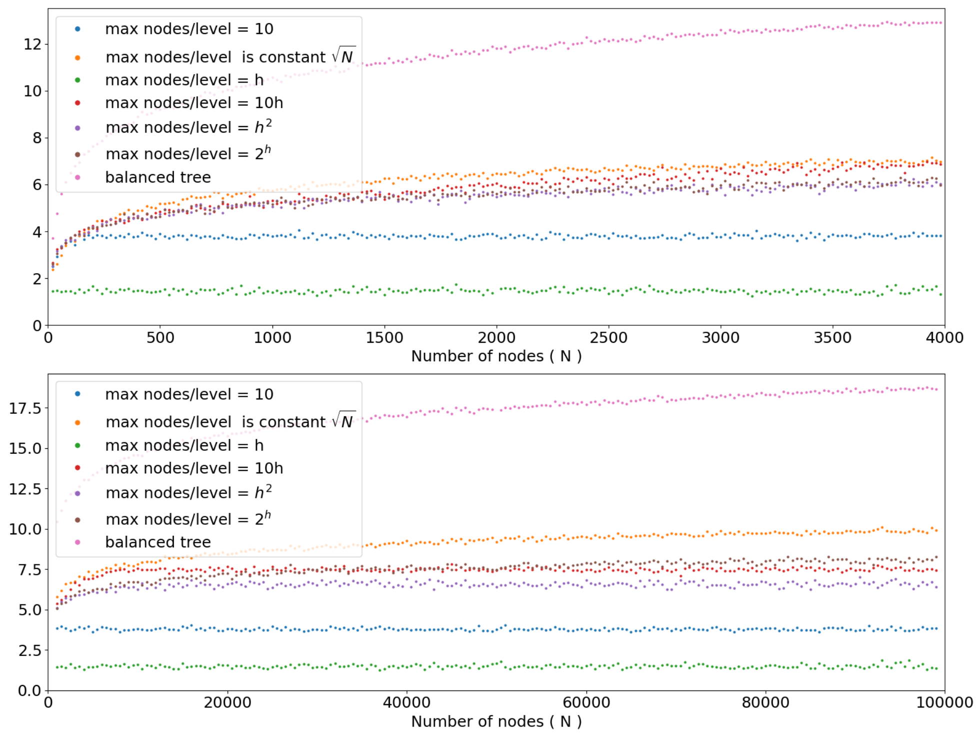


Figure 4: Executing two batches of formula tree generation, above we show the minimum tree depth (level of leaf nodes) for tree sizes up to 4000 nodes, below tree sizes up to 100000 nodes are shown.

In Figure 4 we show the minimum of tree depths as a function of the tree size for different tree shapes. The minimum depth is very important because short-cuts cannot be made without evaluating at least one leaf node.

The maximal depth of expression trees is important as well since there can be threads in the recursive function call flow, where we reach the maximum depth, at which we find the leaf node with the needed value.

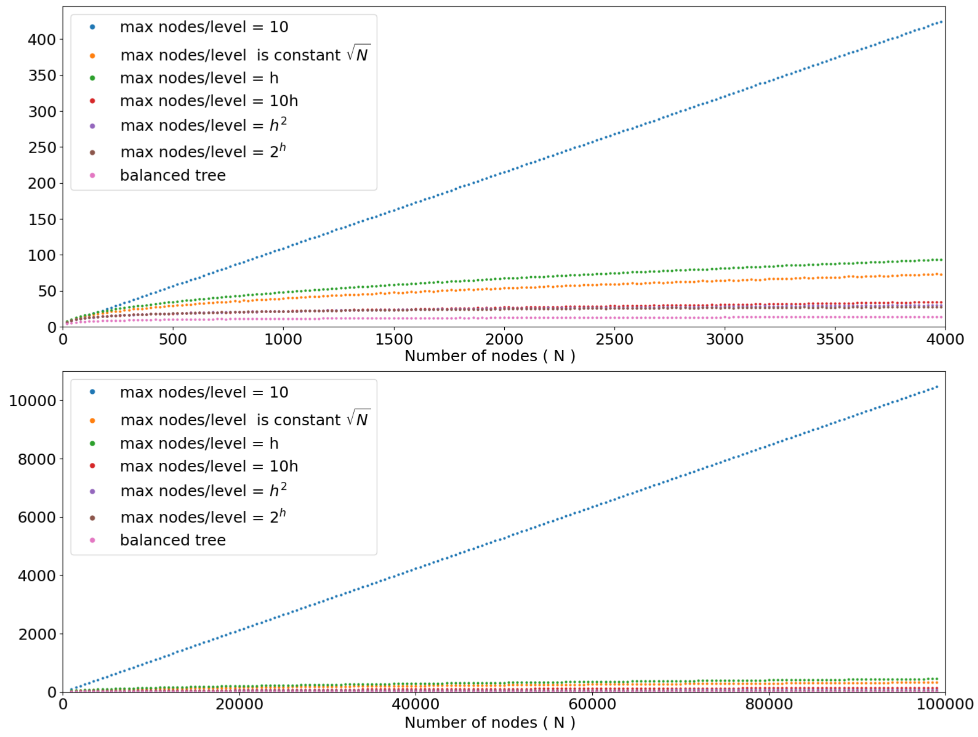


Figure 5: Maximum tree depth, above we show the maximum level of leaf nodes for tree sizes up to 4000 nodes, below sizes up to 100000 nodes are shown.

Figure 5 shows the maximum level of leaf nodes on average for the expression trees generated in various shapes.

As one can see, the narrowest tree shapes have the lowest minimum depths and the highest maximum depths, while the widest shapes on the figure have the lowest maximum depths and their average minimum depths are among the highest ones. There is no contradiction, there can be leaf nodes closer to the root node only if other leaf nodes will be farther.

Also, we must note for the reader that depths on the figures can be not only integral numbers because many experiments were executed with the same parameters including the number of nodes and we show the average results.

5 New fast evaluation algorithms for fuzzy logic formula trees

5.1 Short circuit evaluations in Gödel type fuzzy logic

A research discovering several pruning techniques has been presented in [2]. Although alpha-beta pruning was mentioned there as a possible pruning technique which might be applied for the evaluation of Gödel type fuzzy logic tree [2, 21], the research there was focused on special cases that can occur in a Gödel formula tree where pruning can be done. Such special cases are: implication node with min and max (or max and min) children, implication with negation child, negation when all connected leaves are non-zeros. In this dissertation we present an other solution with a simple but very effective recursive fast evaluation algorithm and its yet better variant.

In this chapter, versions of simple pruning algorithms using lower and upper limits, inspired by alpha-beta pruning [8, 60], are proposed with some simple extensions for the effective evaluation of expression trees in Gödel logic, in which the previously defined operations are used.

We are dealing with trees with a bounded set of truth values: the real numbers of the closed interval $[0, 1]$ can be used. In our expression trees, in a similar manner to the Boolean expressions (actually, the syntax of expressions of Gödel logic is the same as the syntax of Boolean expressions) since both AND and OR are associative, without loss of generality we can allow multiple children of nodes of these types. Nodes with negation must have exactly one child, while nodes with implications must have exactly two children, called a left child and right child, respectively.

The proposed algorithms to accelerate the evaluation of this kind of tree are described in detail in the following.

We will use a technique similar to alpha-beta pruning. Despite the similarity, we will use the terms *lower* and *upper* for the known lower and upper limits of the interval in which in the actual function call the exact value of the node is interesting. If the interval has zero-length then the value of the node can be omitted from the evaluation.

```
function PruneEval1(node, lower, upper):
  if lower >= upper:
    return lower   # Prune

  if NODE_TYPE == NEG:
    v = PruneEval1(child, 0.0, MINPOS)
    if v == 0:
      v = 1.0
    else:
      v = 0.0

  elif NODE_TYPE == MIN:
    v = upper
    for child in children:
      v = min(v, PruneEval1(child, lower, v))
      if v <= lower:
        return lower

  elif NODE_TYPE == MAX:
    v = lower
    for child in children:
      v = max(v, PruneEval1(child, v, upper))
      if v >= upper:
        return upper

  elif NODE_TYPE == IMP:
    leftval = PruneEval1(leftchild, 0.0, upper)
    rightval = PruneEval1(rightchild, min(max(leftval-MINPOS,0.0)
    , lower), min(leftval, upper))

    if leftval <= rightval:
      v = 1.0
    else:
      v = rightval

  else: # NODE_TYPE == LEAF
    v = node value

  return v

end function

value_of_the_expression = PruneEval1(root, 0.0, 1.0)
```

Algorithm 1: PruneEval1 for Gödel fuzzy logic

The function must be called with the root node as the first parameter, 0.0 for *lower* parameter, and 1.0 for *upper* parameter. The evaluation works recursively, it will calculate the value of a node from its children nodes via recursive function calls, making short-cuts where possible. The *lower* parameter, when calling the function for a given node, means that at the evaluation of that node, it does not matter that the value of it is less than *lower* or equal to *lower*. Similarly, the *upper* parameter means, that if we know that the value of the node is at least the value of this parameter, then the exact value is not needed.

Pruning the evaluation, i.e. short-cut, will be made if the value of the *lower* parameter is higher than or equal to the value of the *upper* parameter because it means that the value of that node has no effect on the value of the formula tree.

In case the actual node is a negation, we know that the result of it will be zero if its child node has any value above 0. Therefore we can use an arbitrarily small positive real number which is higher than 0 in the programming environment we are using. We use the $MINPOS$ constant for that purpose, which can represent the smallest float value which is greater than 0. For the *lower* parameter we cannot give a value above 0.0.

For the MIN (AND) nodes we can use the actual *lower* parameter as the *lower* parameter for the function call to evaluate its first child. It is because when the value of the first child will be lower than or equal to the *lower* parameter, then the final result of the actual node cannot be higher than that. We also can use the value of the actual *upper* parameter as the *upper* parameter for the evaluation of the child nodes, because one of the values of the child nodes will be the final result of the evaluation of the actual node and we are not interested in values higher than *upper*. When iterating over the children nodes we also know that the final value of the actual node cannot be greater than any of them, so we can use the minimal known child value or the value of the actual *upper* parameter, which is less, for the upper limit for subsequent function calls. Furthermore, we can cut-off the evaluation of further child nodes if the current known minimal value is less than or equal to the actual *lower* parameter because the value of the actual node cannot be higher than that.

When the type of the actual node is MAX (OR), we can make similar decisions. We can use the actual *upper* parameter as the *upper* parameter for the function call to evaluate its first child. It can be done because if the value of the first child will be at least the value of the *upper* parameter, then the result of the evaluation of the actual node cannot be less than that value. Additionally, we can use the value of the actual *lower* parameter as the *lower* parameter when evaluating its children because values less than that are not interesting. When we evaluate the children nodes we also can use the actual known maximal value of them because the final value of the actual node cannot be less than that. We can prune the evaluation of further child nodes in case the maximal value of the already evaluated nodes is higher than or equal to the actual *upper* parameter. This can be done since the final result of the actual node cannot be lower than that value.

In the case of an IMP node in Algorithm 1 (PruneEval1 for Gödel fuzzy logic), we also use simple pruning. We start the evaluation with the left child node. For this function call, we cannot use the *lower* parameter, because the lower left node values can lead to higher values for the actual node. We can use the actual *upper* parameter for the *upper* parameter of the function call since for the actual node we are not interested in higher values than the value of the *upper* parameter and if the left child node is greater than or equal to that, then we have the following cases:

a) $leftval \leq rightval$: the result is 1 which is higher than or equal to *upper*

b) $leftval > rightval$: the result is $rightval$,

so increasing $leftval$ cannot lead to lower result value.

For the second step in this evaluation of the IMP node, for the right child node, we know that the result of the evaluation of the current node cannot be less than the value of the right child node. We also know that if the right child node has a value greater than or equal to the value of the left child node then the value of the current node will be 1. Therefore, for the *upper* parameter of the function call, we can use the minimum of the current *upper* parameter and the value of the left child node. For the *lower* parameter of the function call to calculate the value of the right child node, we can use the value of the actual *lower* parameter or the value of the left child, which is less, since the right child node can give the value for the actual node if and only if it is less than the value of the left node.

In the following, we present a slightly enhanced version of the algorithm.

```
function PruneEval2(node, lower, upper):
  if lower >= upper:
    return lower # Prune

  if NODE_TYPE == NEG:
    v = PruneEval2(child, 0.0, MINPOS)
    if v == 0:
      v = 1.0
    else:
      v = 0.0

  elif NODE_TYPE == MIN:
    v = upper
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1

    for child in children:
      v = min(v, PruneEval2(child, lower, v))
      if v <= lower:
        break

  elif NODE_TYPE == MAX:
    v = lower
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1
    for child in children:
      v = max(v, PruneEval2(child, v, upper))
      if v >= upper:
        break

  elif NODE_TYPE == IMP:
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      leftval = PruneEval2(leftchild, 0.0, upper)
      rightval = PruneEval2(rightchild, min(max(leftval-MINPOS
    ,0.0), lower), min(leftval, upper))
    else:
      rightval = PruneEval2(rightchild, 0.0, upper)
      leftval = PruneEval2(leftchild, rightval,min(rightval+
   MINPOS,upper))
    if leftval <= rightval:
      v = 1.0
    else:
      v = rightval

  else: # NODE_TYPE == LEAF
    v = node value

  return v

end function

value_of_the_expression = PruneEval2(root, 0.0, 1.0)
```
Algorithm 2: PruneEval2 for Gödel fuzzy logic

Algorithm 2 (PruneEval2 for Gödel fuzzy logic) has similar properties as Algorithm 1, and has additional features.

For the evaluation of the MIN and MAX nodes, we can first check whether the first two child nodes should be processed in reverse order. This is done by checking their node types. If the first child node is not a leaf node and the second child node is a leaf node then the two nodes will be swapped in the row of processing. This is because thus the leaf node will give immediate value when calling the function recursively for its evaluation. Choosing a leaf child node to process first, can lead to an earlier cut-off. We note that it would be possible to change the algorithm to check further node(s) as well when there are more than two child nodes and the first two child nodes are not leaf nodes. This change would have a smaller effect on the probability of the earlier cut-off but would lead to a slower execution because of the additional complexity.

A similar technique has been introduced for the IMP node as well. For this type of node, this cannot be done by a simple swap in the order of processing the child nodes because setting the *lower* and *upper* parameters for the recursive function calls will differ regarding which child node will be processed first. If there will be no swap in the order of processing the child nodes then it will work the same way as in PruneEval1. If the order of the processing of the nodes will be changed then the possible settings of the *lower* and *upper* parameters can be done as follows. When we evaluate the right child node first, we can use the value of the actual *upper* parameter for the *upper* parameter of the function call, because the value of an IMP node is greater than or equal to the value of its right child node. Next, for the evaluation of the left child node, we can use the previously calculated value of the right child node for the *lower* parameter because if the value of the left node is less than or equal to the value of the right child node, the result of the actual node will be 1. For the *upper* parameter we can use the value of the actual *upper* parameter, when it is not greater than the previously calculated value of the right child node.

5.2 Algorithm with short-cuts in Product fuzzy logic

We will now discuss some short-cut possibilities which can be used to evaluate Product logic formula trees [25, 26, 3]. We are working with formula trees of different shapes and different value distributions. These properties have an effect on the possible efficiency of the pruning evaluation. The shape of the formula tree determines the minimal, mean, and maximal depth of the tree. The minimal depth of the tree determines the minimal number of processed nodes before a short-cut can be made. The algorithm gets the formula, represented by an expression tree. The vertices have a type and connections (edges) to its children nodes, while the leaf nodes have their real values from the closed interval $[0, 1]$. The evaluation starts at the root node by calling the recursive evaluation function, and it will compute the value at the root, recursively. For the Product logic formula tree, the negation nodes have exactly one child, while the nodes of the AND, OR, and IMP operators have two child nodes. Although in product logic the algorithm can be easily extended to allow more than two child nodes for the AND and the OR nodes, in this implementation we will not use more than two child nodes for them.

In the following, we include two versions of the new algorithm with explanations of the major possible pruning possibilities.

```
function PruneEval1(node, lower, upper):
  if lower >= upper:
    return lower  # Prune

  if NODE_TYPE == NODE_TYPE_NEG:
    v = 1.0 - PruneEval1(child, 1.0-upper, 1.0-lower)

  elif NODE_TYPE == NODE_TYPE_AND:
    v = PruneEval1(child1, lower, 1.0)
    if v > lower:
      v = v * PruneEval1(child2, lower, 1.0)

  elif NODE_TYPE == NODE_TYPE_OR:
    v = PruneEval1(child1, 0.0, upper)
    if v < upper:
      B = PruneEval1(child2, 0.0, upper)
      v = v + B - v * B

  elif NODE_TYPE == NODE_TYPE_IMP:
    A = PruneEval1(child1, 0.0, 1.0)
    if A == 0.0:
      v = 1.0
    else:
      B = PruneEval1(child2, 0.0, A)
      v = min( B / A, 1.0)

  else: # NODE_TYPE_LEAF
    v = nv

  return v
end function

value_of_the_expression = PruneEval1(root, 0.0, 1.0)
```

Algorithm 3: PruneEval1 for Product fuzzy logic

Algorithm 3 has only the simplest short-cut techniques for the Product logic formula evaluation implemented. The algorithm works as follows. It must be started with the root node as a parameter. For the support of the pruning a *lower* and an *upper* parameter are presented, which mean the boundaries of the interval in which we are interested in the exact value of the node, that means, that if the value of the node falls below the *lower* parameter, then this is enough information for us, similarly, if the value of the node is higher than or equal to the value of the *upper* parameter, then we don't need to know the exact value of it.

The short-cut can be done simply when the *lower* has a value higher than or equal to the value of the *upper* parameter because in that case, the interesting interval has zero length.

In the case of the negation node, the setting of the values of the *lower* and *upper* parameters of the function call for the evaluation of the child node of the negation node can be done by simply mirroring in respect of the interval, i.e. subtracted from 1.

For the AND nodes we know that it is commutative and if either node has already its calculated value, then the result of the AND operation cannot be higher than that value. Therefore for the calculation of either node, we can use the value of the actual *lower* parameter for the *lower* parameter of the function call, and can stop the evaluation without calling the function for the second child node if we already know that the value of this node will be less than or equal to *lower*.

The OR nodes have a similar possibility to cut-off the unnecessary nodes from the evaluation. The result of the OR operation will be higher than or equal to the value of either of its child nodes. Therefore we can use the value of the actual *upper* parameter for evaluating both of its children, as the *upper* parameter of the function call. Also we can stop the evaluation before the second child node if the first child node had a value at least *upper*.

In the case of IMP node, the first function call to evaluate their children must be called with *lower* $= 0$ and *upper* $= 1$ parameters since the result of the operation can be any value between 0 and 1. If we already got the value of the left child node, it can be used as the *upper* parameter of the function call to compute the value of the right child node because if the value of the right child node is higher than or equal to the value of the left node, then the result of the operation will be 1.

An improved variant of the presented pruning algorithm will also be shown.

```
function PruneEval2(node, lower, upper):
  if lower >= upper:
    return lower  # Prune

  if NODE_TYPE == NODE_TYPE_NEG:
    v = 1.0 - PruneEval2(child, 1.0-upper, 1.0-lower)

  elif NODE_TYPE == NODE_TYPE_AND:
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1
    A = PruneEval2(child1, lower, 1.0)
    if A == 0.0:
      v = 0.0
    else:
      v = A * PruneEval2(child2, min(lower/A, 1.0),
                min(upper/A, 1.0))


  elif NODE_TYPE == NODE_TYPE_OR:
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1
    A = PruneEval2(child1, 0.0, upper)
    if A == 1.0:
      v = 1.0
    else:
      B = PruneEval2(child2, max((lower-A)/(1.0-A),0.0),
              (upper-A)/(1.0-A))
      v = A + B - A * B

  elif NODE_TYPE == NODE_TYPE_IMP:
    if child1type==NODE_TYPE_LEAF or child2type!=NODE_TYPE_LEAF:
      A = PruneEval2(child1, 0.0, 1.0)
      if A == 0.0:
        v = 1.0
      else:
        B = PruneEval2(child2, A*lower, A*upper)
        v = min(B/A,1.0)
    else:
      B = PruneEval2(child2, 0.0, 1.0)
      if B == 1.0:
        v = 1.0
      else:
        A = PruneEval2(child1,min(B/upper,1.0),
              1.0 if B==0 else
              min(1.0 if lower==0 else B/lower,1.0))
        if A == 0.0:
          v = 1.0
        else:
          v = min(B/A,1.0)

  else: # NODE_TYPE_LEAF
    v = nv

  return v
end function

value_of_the_expression = PruneEval2(root, 0.0, 1.0)
```

Algorithm 4: PruneEval2 for Product fuzzy logic

Algorithm 4 has all the pruning capabilities as the variant Algorithm 3, and it has been further developed to include more cut-off possibilities.

For the negation nodes, there has been no new pruning technique introduced.

In the case of an AND node, it can swap the order of the evaluation of its child nodes if the first child is not a leaf node and the second child is a leaf node. It will improve the efficiency because at a leaf node we know that there are no further nodes below it, so it will give an immediate value, which then might possibly lead to a short-cut. After the order of the processing of the child nodes has been determined, the first child node will be evaluated the same way, no new pruning technique added here. For the recursive call to evaluate the second child node, however, we introduce improvements for setting the *lower* and *upper* parameters for the function call. Instead of using 0.0 for *lower*, now we set it to $lower/A$ if it is less than 1.0, otherwise, we use 1.0. This can be done because the value of the node will be a factor in the multiplication. We also can use a better limit for the *upper* parameter of the function call. This will be $upper/A$ if it is less than 1.0 otherwise it will be 1.0.

For the OR nodes, we also have improved the pruning capability. Similar to the AND nodes we introduced the swapping ability in the order of the evaluation of its child nodes. For the evaluation of the first child, the method given in algorithm PruneEval1 for Product logic has been kept unchanged. At the evaluation of the second child node, we have made an improvement to the algorithm. Instead of having only one pruning condition, which was the usage of the actual *upper* parameter for the *upper* parameter of the function call, we can set a value for it which is less than or equal to that, so it can lead to mode short-cuts. This value will be $(upper - A)/(1.0 - A)$, which comes for FB from the inequality

$$A + B - A * B > upper.$$

For the *lower* parameter of the function call, we also can now use the already calculated value of the first child node. From the inequality

$$A + B - A * B < lower$$

we get the $(lower - A)/(1.0 - A)$ for B, which we can use there for parameter *lower* if it is higher than or equal to 0.0, otherwise we use 0.0.

We also have improvements for the implication nodes. We start by determining the order of the evaluation of the child nodes by checking their types. We make note that only the processing order of the child nodes can be changed here, it is not allowed to swap them because the IMP operation is not commutative. For the first recursive function call, we cannot make a restriction for the *lower* and *upper* bounds. If we started with the left child node then we can use its value multiplied by *lower* for the *lower* parameter, and multiplied by *upper* for the *upper* parameter for the function call. On the other branch, when we have started the calculation of the IMP node with the right child node, then we can use $B/upper$ for the *lower* parameter and $B/lower$ for the *upper* parameter of the recursive function call to evaluate the left child node.

5.3 Fast evaluation algorithms for Łukasiewicz fuzzy logic

We use expression trees with the set of truth values from the real numbers of the closed interval $[0, 1]$, they can be used on the nodes of the tree. They are given on the leaves of the tree at the beginning of the evaluation, and the task is to calculate the value at the root. Trees are unary-binary in the sense that each vertex has a maximum of two children: the vertices assigned to the negation must have exactly one child, while the other vertices with assigned (binary) connections must have exactly two children, the so-called left child (child1) and the right child (child2).

We have written the algorithms by recursive pseudo-codes, see e.g. Algorithm 5. Given the input formula tree with its root and two parameters, *lower* and *upper*, the algorithms compute the value y of the expression if $lower \leq y \leq upper$. The interesting interval is represented by the numbers *lower* and *upper*, as its lower and upper limits, for the value v of a node such that this value has an influence on the result of the main formula, if v is in its interesting interval. This interval depends on the already analyzed part of the expression(tree). The cuts during the evaluation are performed by adjusting the parameters *lower* and *upper* dynamically during the run for each node of the formula tree. More precisely, the role of the parameters *lower* and *upper* is as follows.

While evaluating the children of the current node we must continue the evaluation only if their value can be between the actual limits *lower* and *upper* provided for their evaluation. Intuitively, when $lower >= upper$, there is no interval between them, thus the given node or sub-tree can be cut, the value of the main formula does not depend on the value(s) of this node or sub-tree. We do the recursive evaluation of the formula tree with the simple recursive function call. The main call can be seen in the last line of the Algorithm 5.

Below we show the recursive pseudo-code for pruning evaluation of expressions in Łukasiewicz logic.

```
function PruneEval1(node, lower, upper):
  if lower >= upper:
    return v = lower # cut

  if nodetype == NODE_TYPE_NEG:
    v = 1 - PruneEval1(child, 1-upper, 1-lower)

  elif nodetype == NODE_TYPE_AND:
    v1 = PruneEval1(child1, lower, 1)
    v2 = PruneEval1(child2, min(1+lower-v1,1), min(1+upper-v1,1))
    v  = max(v1+v2-1, 0)

  elif nodetype == NODE_TYPE_OR:
    v1 = PruneEval1(child1, 0, upper)
    v2 = PruneEval1(child2, max(lower-v1,0), max(upper-v1,0))
    v  = min(v1+v2, 1)

  elif nodetype == NODE_TYPE_IMP:
    v1 = PruneEval1(child1, 1-upper, 1)
    v2 = PruneEval1(child2, max(lower-1+v1,0), max(upper-1+v1,0))
    v  = min(1-v1+v2, 1)

  else: # NODE_TYPE_LEAF
    v = node value

  return v
end function

value_of_the_expression = PruneEval1(root, 0.0, 1.0)
```

Algorithm 5: PruneEval1 for Łukasiewicz fuzzy logic

The negation node will mirror the $[0, 1]$ interval, so the limits *lower* and *upper* must also be transformed to $1 - \mathit{upper}$ and $1 - \mathit{lower}$ for the evaluation of its child node.

There are two special possibilities to prune at a conjunction (AND) node. The first is, that if the value of its first child is less than or equal to lower, then we do not need to evaluate the second child, because the value of the conjunction node cannot be higher than the value of any of its children. The second pruning possibility is that the value of the first child can be used to set stronger limits for the evaluation of the second child.

There are also two special possibilities for pruning at a disjunction (OR) node. The first one is that if the value of the first child is at least the upper limit, then the precise value of it is not needed because the result of the disjunction node cannot have a value less than the value of any of its children. The second pruning possibility is that the value of the first child can be used to set tighter limits for the evaluation of the second child.

For evaluating the implication (IMP) node, at least one of its children must be evaluated because until then any value from the $[0, 1]$ interval is possible. For simplicity, in Algorithm 5 we always start with the evaluation of the first child. The value of it can then be used to set the limits *lower* and *upper* for the evaluation of the second child.

The classical short-circuit evaluation techniques are also working and they are encoded in the algorithm in the following way.

When evaluating a conjunction node, if its first child has value $v_1 = 0$, then the evaluation of the second child is called with $lower = 1$ and $upper = 1$, resulting short-cut for that child, the value of the conjunction node will be set to 0 (by the statement $v = max(v_1 + v_2 - 1, 0)$, independently of the value $0 \leq v_2 \leq 1$).

Similarly, at a disjunction node, having a child with value $v_1 = 1$, the evaluation of the other child is called with $lower = upper = 0$ implying an automatic cut for this child. Moreover, the value 1 is assigned to the disjunction node as $v = min(1 + v_2, 1)$ independently of the actual value of v_2.

Because of symmetry, for the first view, there is no real reason why the evaluation should break the usual left to right order. This assumption holds without taking into account the sizes of the branches, i.e., the sizes of the left and right sub-formulae. On average, for random formulae, the sizes of the left and right sub-formulae are equal. Algorithm 5 uses the traditional left to right evaluation. Intuitively one can feel that the evaluation process could be faster if the shorter branches are evaluated first. We have applied a similar idea also in [2, 3]. Here we can also use it as follows: if the right child of the node is a leaf, but the left child is not a leaf, then we evaluate the right child earlier than the left child. The two children of conjunction and disjunction nodes can be checked for their type, to evaluate the leaf node first if there is one. This improvement gives better pruning ratios and evaluation times. In the code, this can be done, e.g., in the following way, by simply interchanging them.

```
      if child1type!=NODE_TYPE_LEAF and
         child2type==NODE_TYPE_LEAF:
        child1,child2=child2,child1
```

A similar trick can be done at implication nodes taking into account that implications are not commutative. Algorithm 6 has only one additional feature comparing it to Algorithm 5: the leaf checking. At binary operations, i.e., at conjunctions, disjunctions, and implications, if the left child is not a leaf, but the right child is a leaf, then we break the left to right order of the evaluation, we start first by evaluating the right child, the leaf by taking its value, and we can already use this value to have a stronger constraint (i.e., a smaller interesting interval) to evaluate the other, the left child.

Next, we show the recursive code for pruning by evaluating the right child leaf node first if the left child is non-leaf.

```
function PruneEval2(node, lower, upper):
  if lower == upper:
    return v = lower # cut

  if nodetype == NODE_TYPE_NEG:
    v = 1-PruneEval2(child, 1-upper, 1-lower)

  elif nodetype == NODE_TYPE_AND:
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1
    v1 = PruneEval2(child1,lower,1)
    v2 = PruneEval2(child2,min(1+lower-v1,1),min(1+upper-v1,1))
    v  = max(v1+v2-1,0)

  elif nodetype == NODE_TYPE_OR:
    if child1type!=NODE_TYPE_LEAF and child2type==NODE_TYPE_LEAF:
      child1,child2=child2,child1
    v1 = PruneEval2(child1,0,upper)
    v2 = PruneEval2(child2,max(lower-v1,0), max(upper-v1,0))
    v  = min(v1+v2, 1)

  elif nodetype == NODE_TYPE_IMP:
    if child1type==NODE_TYPE_LEAF or child2type!=NODE_TYPE_LEAF:
      v1 = PruneEval2(child1,1-upper, 1)
      v2 = PruneEval2(child2,max(lower-1+v1,0),max(upper-1+v1,0))
    else:
      v2 = PruneEval2(child2,0, upper)
      v1 = PruneEval2(child1,min(1+v2-upper,1),min(1+v2-lower,1))
    v = min(1-v1+v2,1)

  else: # NODE_TYPE_LEAF
    v = node value
  return v
end function

value_of_the_expression = PruneEval2(root, 0.0, 1.0)
```

Algorithm 6: PruneEval2 for Łukasiewicz fuzzy logic

This improved version includes the ability to change the order of evaluation of the child nodes for the IMP node as well, although this operator is not commutative. To do this, it must change the setting of the *lower* and *upper* parameters for the function calls evaluating the child nodes, when we evaluate the right child node first if it is a leaf node and the left node is non-leaf.

5.3.1 Proof of correctness of the algorithms [4]:

Now we are proving the correctness of our algorithms. Some parts of the two algorithms are very similar, or even identical, thus some parts of the proof can be considered for both algorithms. While we use the variables *lower* and *upper* for the current node, for its child(ren) we use the variables $lower_1$ and $upper_1$ and $lower_2$ and $upper_2$ (for the second child if exists) in the proof (see Appendix). We start the proofs with a simple but important observation written in the following lemma.

Lemma 1. Both Prune1 and Prune2 use variables *lower*, *upper* and v such that they always have values with condition $0 \leq lower \leq 1, 0 \leq upper \leq 1, 0 \leq v \leq 1$ during the recursive calls.

The proofs of this and the next lemmas are moved to the Appendix for better readability of the paper.

Lemma 2. Each time Algorithm 5 (PruneEval1 for Łukasiewicz logic) and Algorithm 6 (PruneEval2 for Łukasiewicz logic) is called recursively if the condition $0 \leq lower \leq upper \leq 1$ is fulfilled, then this will also hold for the parameters of the subsequent recursive call(s).

The next lemma presents a trivial fact, but it plays an important role in the proof by induction later on.

Lemma 3. For any expression having value x, if Algorithm 5 (Algorithm 6, resp.) assigns the correct value x to it, then one of the following statements is fulfilled.

a) The parameters have the relation $lower \leq x \leq upper$, and thus the assigned value is also between *lower* and *upper* (inclusively, allowing equality also).

b) If $x < lower$, then the algorithm assigns a value that is not larger than lower.

c) If $x > upper$, then the algorithm assigns a value that is not less than upper.

Now, let us analyze the special cases when $lower = upper$.

Lemma 4. For any expression having value x, if Algorithm 5 (Algorithm 6, resp.) is called with parameters $lower = upper$, then it assigns

a) the value x of the expression correctly if $lower = x = upper$,

b) a value not larger than *lower* if the value of the formula x is less than *lower*, and

c) a value that is at least upper if the value of the formula x is greater than upper.

By Lemma 2, it is clear that $lower \leq upper$ in each recursive call, and we have already seen the case of equality (Lemma 4). In the next lemmas, we consider the case when $lower < upper$.

Lemma 5. For any expression having value x, if $lower < upper$, then Algorithm 5 assigns

a) the value x of the expression correctly if $lower \leq x \leq upper$,

b) a value not larger than *lower* if the value of the formula x is less than *lower*, and

c) a value that is at least *upper* if the value of the formula x is greater than *upper*.

Let us state analogous statement for Algorithm 6.

Lemma 6. For any expression having value x, if $lower < upper$, then Algorithm 6 assigns

a) the value x of the expression correctly if $lower \leq x \leq upper$,

b) a value not larger than lower if the value of the formula x is less than lower, and

c) a value that is at least *upper* if the value of the formula x is greater than *upper*.

Theorem 1. Algorithm 5 is correct: PruneEval1 evaluates correctly the input formula.

Proof. We have just shown in Lemma 5 that the algorithm gives the value x of the expression correctly if it is called with parameters satisfying $lower \leq x \leq upper$. Initially, calling the algorithm with $lower = 0$ and $upper = 1$ give the sufficient condition that the value of the expression is correctly computed.
Theorem 2. Algorithm 6 is correct: it evaluates correctly the input formula.
Proof. It is analogous to the proof of Theorem 1, by applying Lemma 6.

6 Simulation results of fast evaluation of fuzzy logic formula trees

The simple evaluation algorithm must process each node of the expression tree exactly once, therefore its execution time is linear to the number of the nodes of the tree. The pruning algorithm variants we developed, do not usually evaluate or even do not touch all nodes, so the execution time is usually below linear to the number of nodes, depending on parameters like the formula shape or the distribution of the values of the leaf nodes. We note that comparing execution times of the algorithm variations can be misleading because our experiment was executed by collecting information and that has an overhead on the performance. For small expressions, it can require relatively more time to evaluate. Its advantage can be manifested when it is executed on large formula trees. The algorithm has been developed in the Python language and executed 1,000,000 tests on formulae generated with different parameters, e.g. with various sizes, tree shapes, and value distributions.

Since one of the major goals with optimization is to reach faster execution, we start with a comparison of the execution times. Note that the execution times shown were measured in a multitasking environment and should not be considered accurate.

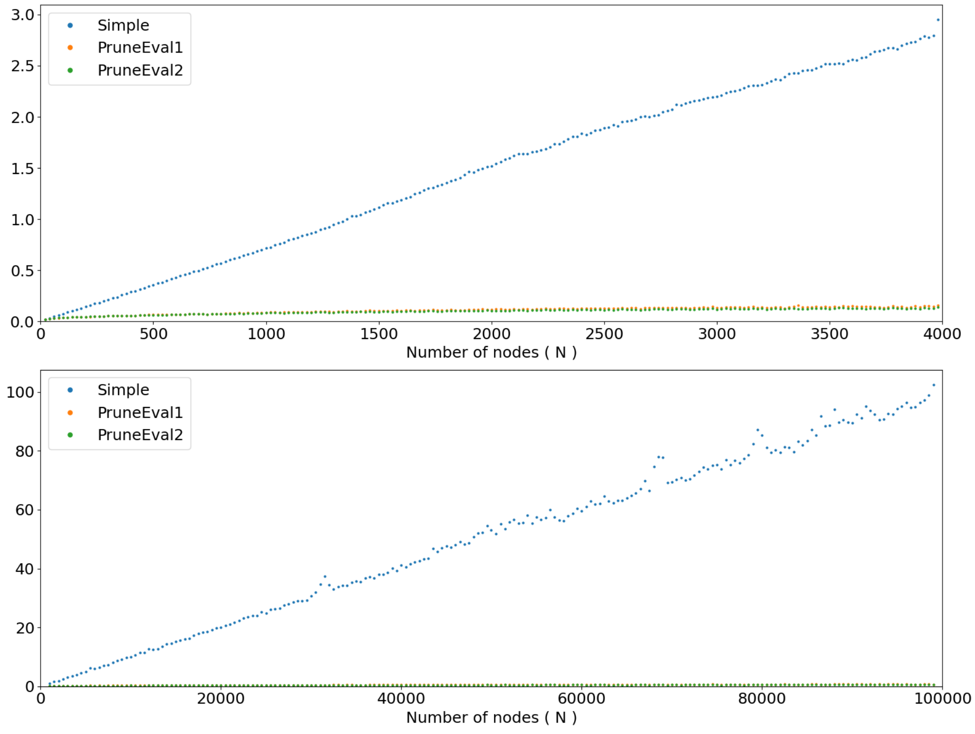


Figure 6: The execution times (in milliseconds) of simple evaluation and the pruning algorithms when run on Gödel fuzzy logic formula trees. See the exec time for tree sizes up to 4000 nodes above and up to 100000 nodes below.

Figure 6 shows the execution time in milliseconds of the pruning algorithms executed on Gödel fuzzy logic formula trees, and also the execution time of the simple evaluation. Our pruning algorithms performed much better, even with smaller formula trees. Their execution times are only a fraction of the execution time of the simple evaluation, the difference grows when we increase the tree sizes.

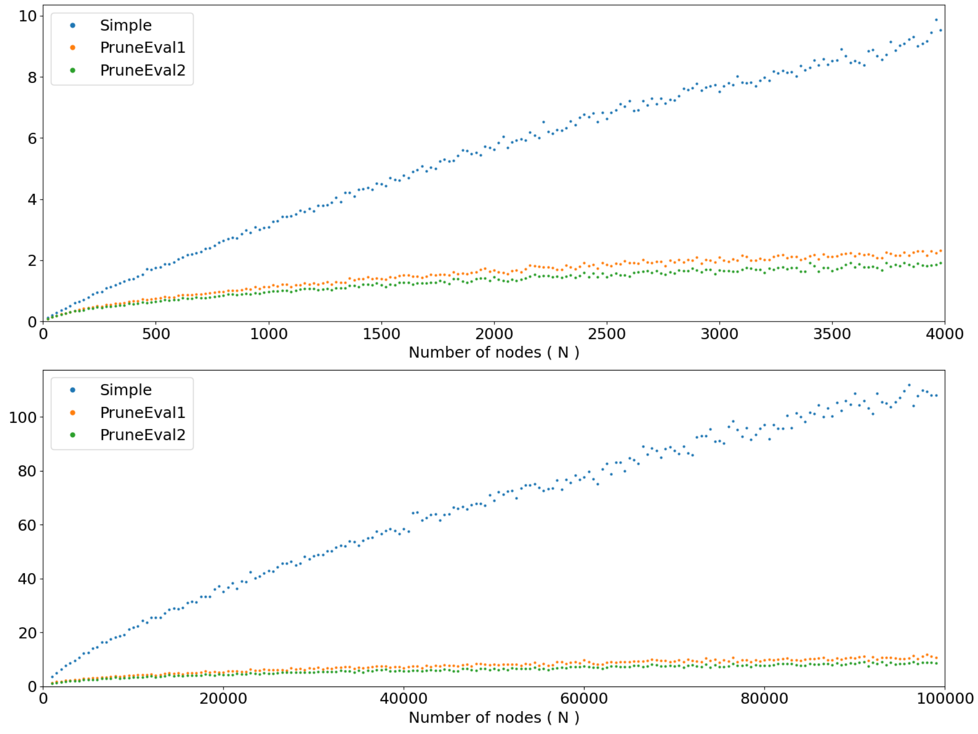


Figure 7: The execution times (in milliseconds) of simple evaluation and the pruning algorithms when for Product logic expressions. See the exec time for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

In Figure 7 we present the simulation results of the executions of the algorithms on Product logic. The difference is significant compared to the case of Gödel logic, the execution times for Product logic are much higher. This is a consequence mainly of the characteristics of the negation operator in Gödel logic because the result of a negation operation in Gödel logic can only be 0 or 1, which leads to more pruning possibilities. If we compare the pruning algorithms vs. the simple evaluation, we can see that the execution time of the simple evaluation is an almost linear function of the number of nodes of the expression trees for the simple evaluation algorithm, and it below linear using the pruning algorithm.

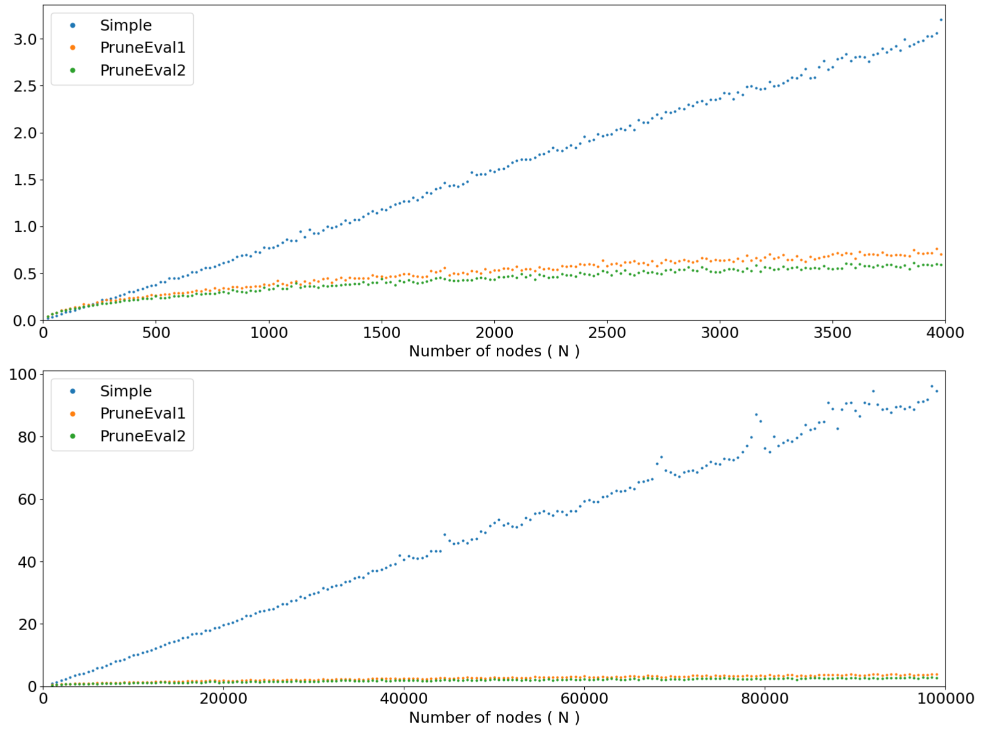


Figure 8: The execution times (in milliseconds) of simple evaluation and the pruning algorithms when run on Łukasiewicz logic formula trees. See the exec time for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 8 presents the execution times of the evaluation of expression trees in Łukasiewicz fuzzy logic. These results are between the results of the execution times on Gödel logic and Product logic expression trees. We already mentioned the characteristics of the negation operator of the Gödel fuzzy logic with resulting only 0 or 1 helps the pruning algorithm. The reason why there is a better pruning possibility of the Łukasiewicz pruning algorithm compared to the Product logic lies in the conjunction and disjunction operators, i.e. the conjunction gives 0 with higher probability and the disjunction gives 1 with a higher probability.

The execution time may vary depending on implementation and machine performance, and also we have used information gathering from the algorithms which affects the execution time, so we give some more evidence of our algorithms' efficiency.

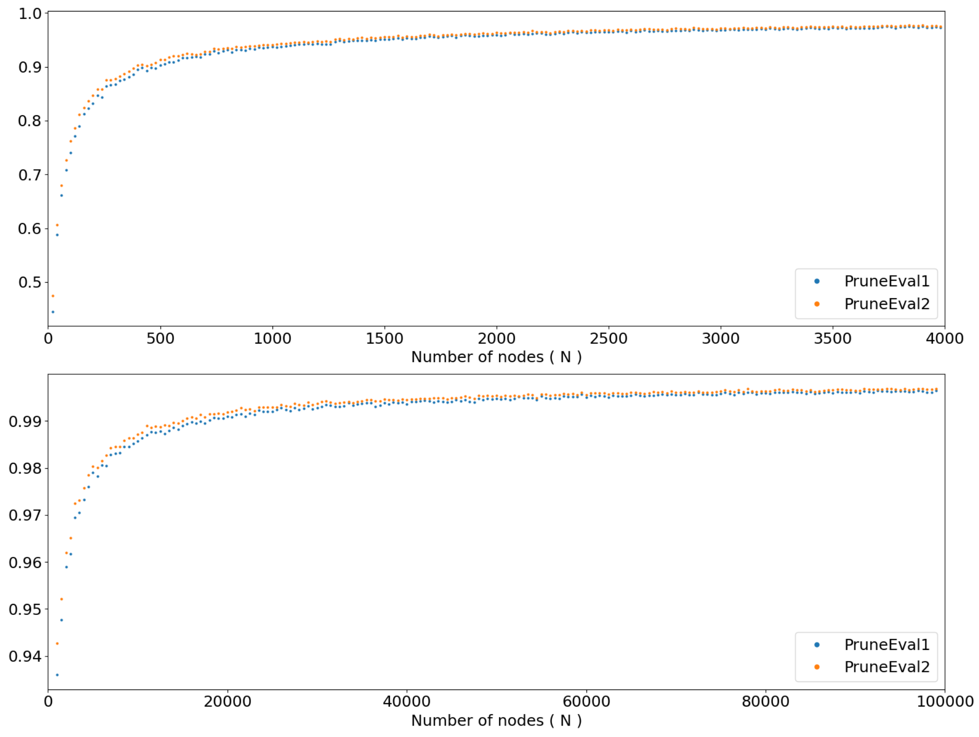


Figure 9: The ratio of the pruned nodes executed on Gödel type fuzzy logic expression trees. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 9 shows the ratio of pruned nodes. For the Gödel type logic the pruning ratio is very high. The ratio of the not evaluated nodes approaches 1 as the number of nodes increases. This very high efficiency is due in part to the fact that the operator of the denial operates in a special way in that type of fuzzy logic, as we discussed earlier.

Because of this behavior, we show below the same figure executed also on Gödel fuzzy logic formulas, but this time the negation was replaced to the same as in Product fuzzy logic.

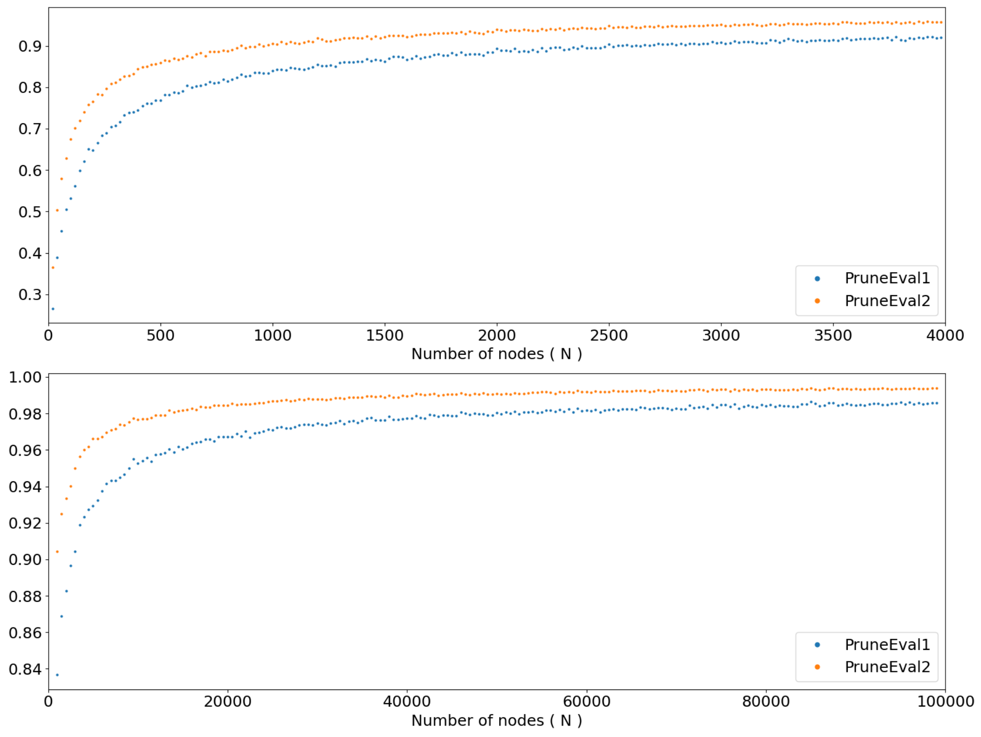


Figure 10: The ratio of the pruned nodes evaluating on Gödel type fuzzy logic expression trees with using negation nodes changed to the one in Product fuzzy logic. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

For comparison, Figure 10 shows the results of a similar experiment for expression trees on Gödel type fuzzy logic with negation nodes temporarily changed to the one in the Product fuzzy logic. The difference is apparent, the simpler version of the pruning algorithm has a lower efficiency than the more advanced one and both pruning algorithms can work with lower efficiency compared to the original case, when there were much more of 0 and 1 values in the formula tree during evaluation because of the behavior of the negation operator.

The proportion of non-evaluated nodes increases with the number of nodes, although not as much as in the previous case.

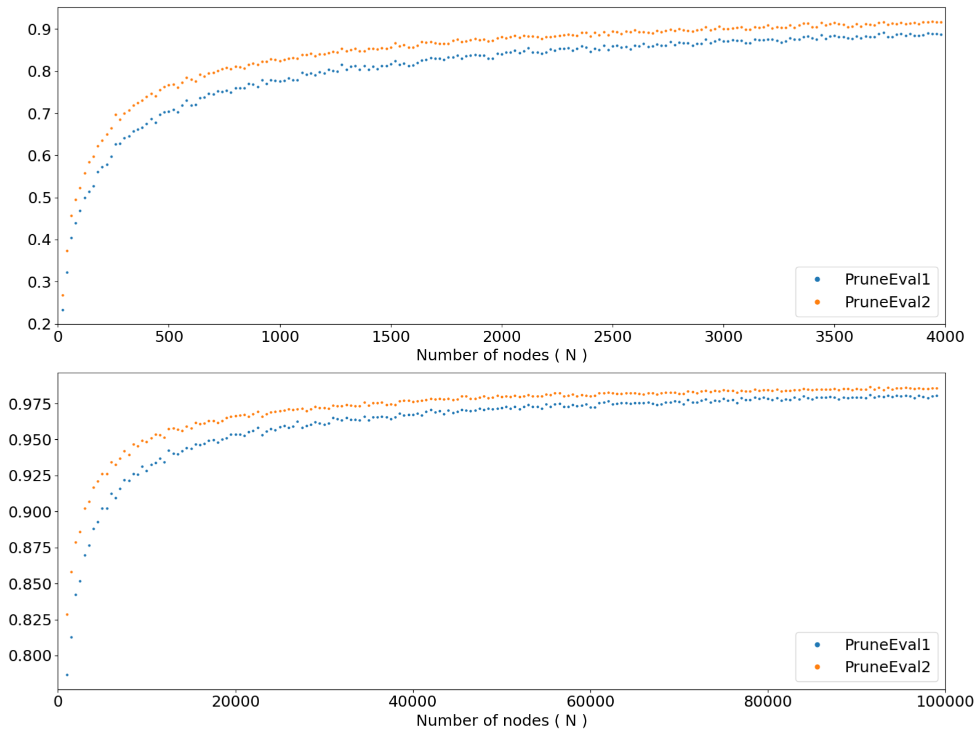


Figure 11: The ratio of the pruned nodes executed on Product type fuzzy logic expression trees. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 11 presents the ratio of the nodes not evaluated by the pruning algorithms on Product logic expression trees. As we might have expected from the comparison of the execution times, this ratio is lower, compared to the results of the evaluation on Gödel type logic formulas because of the different behavior of the negation operator.

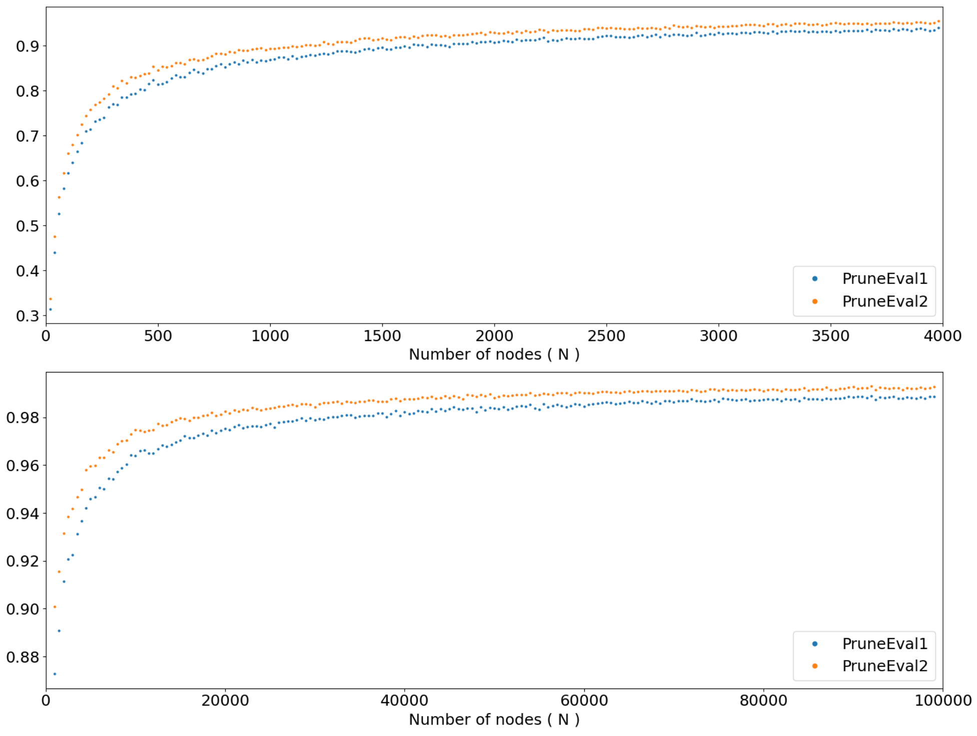


Figure 12: The ratio of the pruned nodes executed on Łukasiewicz type fuzzy logic expression trees. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below

Figure 12 shows the ratio of the nodes was omitted from the evaluation by the pruning algorithms on Łukasiewicz logic formula tree. As for the execution times, this ratio is between the pruned node ratios of the Product logic and the Gödel logic. The reason is the same, as described for the execution time measurement.

The pruning ratio can be different for the various trees with different properties. Now we will see how the value distribution affects the pruning ratio.

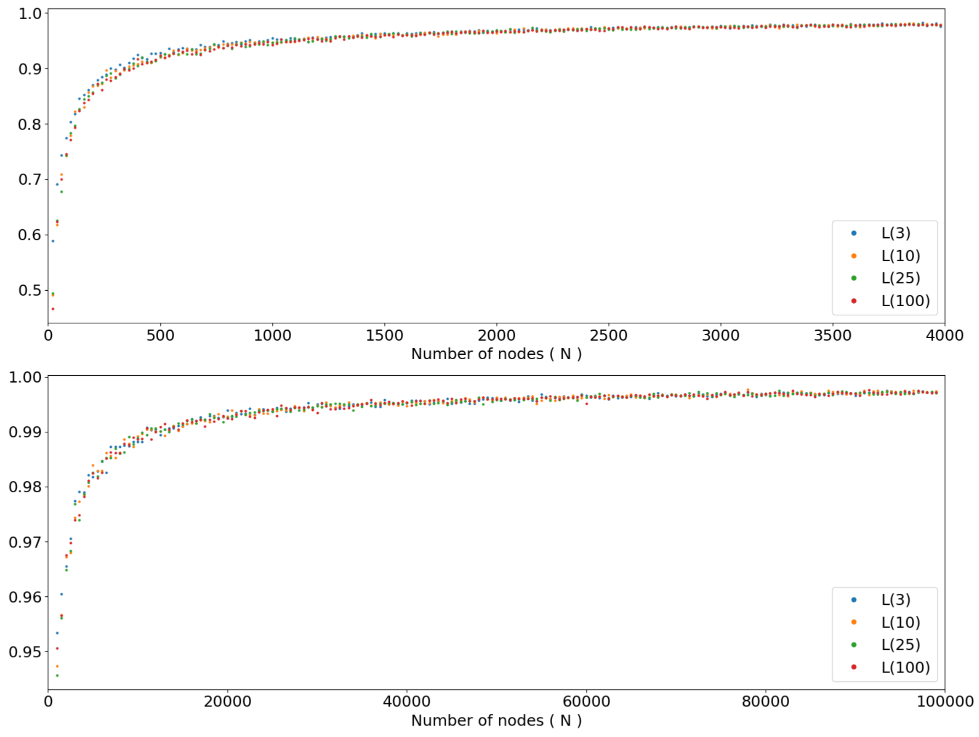


Figure 13: The ratio of nodes of Gödel formula tree pruned by the PruneEval2 algorithm for the Gödel logic, by value distributions. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 13 gives more detailed information on the ratio of the not evaluated nodes for Gödel fuzzy logic expressions by value distributions. As we can see, the ratio of pruned nodes is higher on average with fewer values, and it is less if we have a larger set of values. In the case of L(3) there is a significant (2/3) probability that the value of a leaf node will be 0 or 1. Besides the probabilities of 0 and 1, which easily can lead to pruning, the probability of the equality of two values is also higher, which also can lead to pruning.

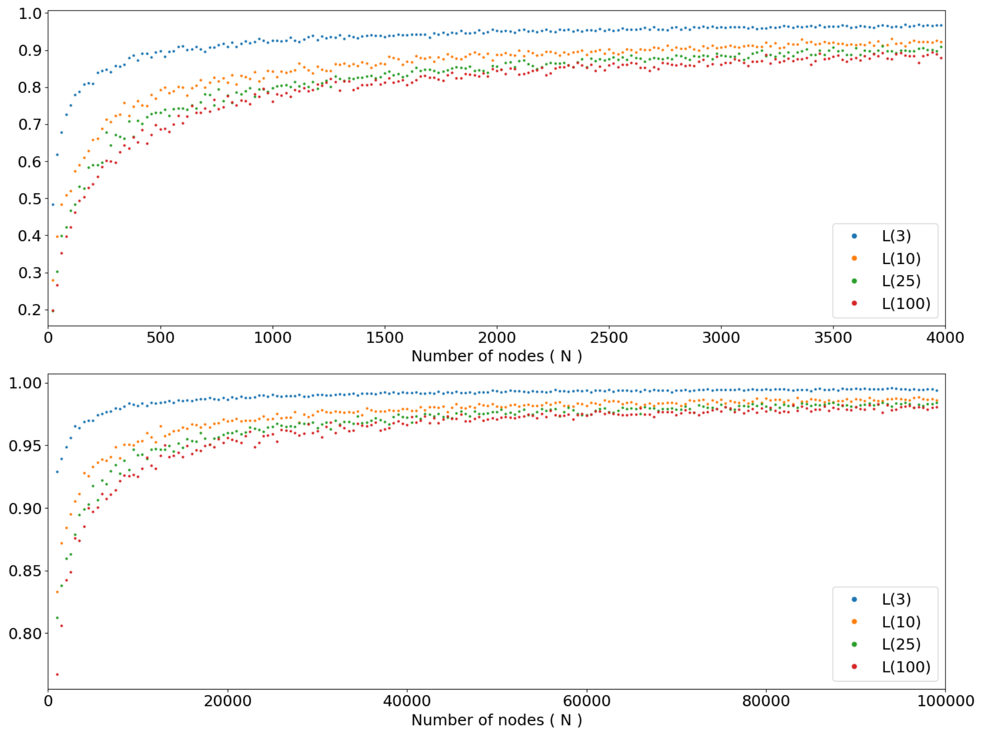


Figure 14: The ratio of nodes of Product formula tree pruned by the PruneEval2 algorithm for the Product logic, by value distributions. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 14 shows the ratio of the not evaluated nodes for Product fuzzy logic expressions for the different value distributions we used. The proportion of pruned nodes is different for the various value distributions. As we can see, the ratio is higher if fewer values were used, and lower if we have a larger set of values. The difference between the L(3) and L(100) cases is much larger than in the case of Gödel fuzzy logic. This is because the Gödel logic had the effect that the negation operator reduces the number of values other than 0 and 1.

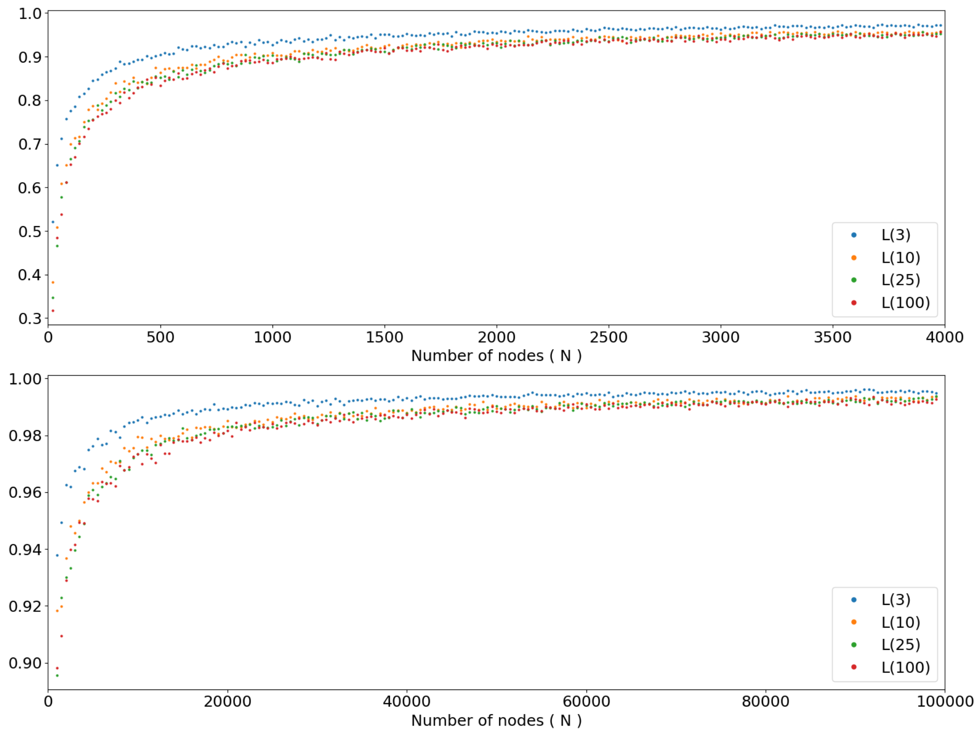


Figure 15: The ratio of nodes of Łukasiewicz formula tree pruned by the PruneEval2 algorithm for the Łukasiewicz logic, by value distributions. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 15 gives more detailed information on the ratio of the not evaluated nodes for Łukasiewicz fuzzy logic expressions by value distributions. As we can see, the ratio of pruned nodes is higher than for the Product logic experiment but lower than compared to the Gödel logic prune ratios. Also, there is a difference in the pruning ratios for value distributions with less or more values.

We will also check the number of the evaluated nodes because the evaluated nodes require performance during the evaluation. We also include the performance of the evaluation using the trivial short-cut possibilities that are used for the evaluation of binary logic formulas.

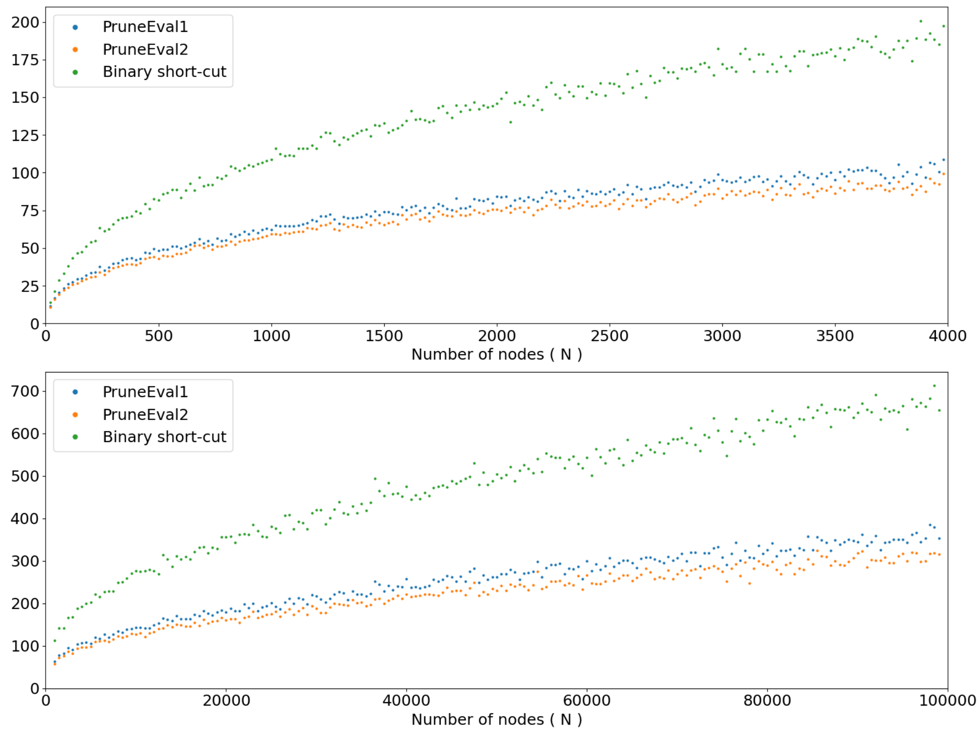


Figure 16: The number of not pruned nodes of Gödel fuzzy logic expression tree. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 16 shows how many nodes of Gödel logic formula tree have been evaluated. Both of our algorithms have done short-cuts very efficiently. The count of the not pruned nodes is below 100 up to tree sizes of 4000 nodes, and just a few hundreds when closing to the large tree sizes of 100k nodes. We can see that using our algorithms only about 50% of the nodes had to be checked compared to the algorithm which used the short-cuts usual in binary logic (e.g. if the first operand of the AND operation is 0 then we don't need to evaluate the second operand).

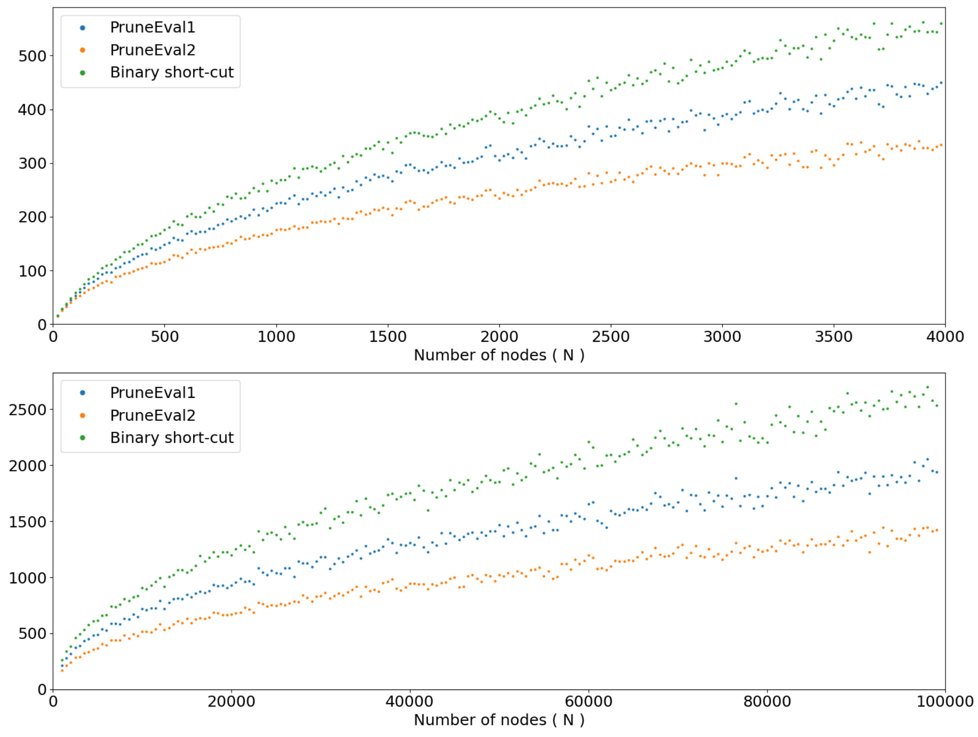
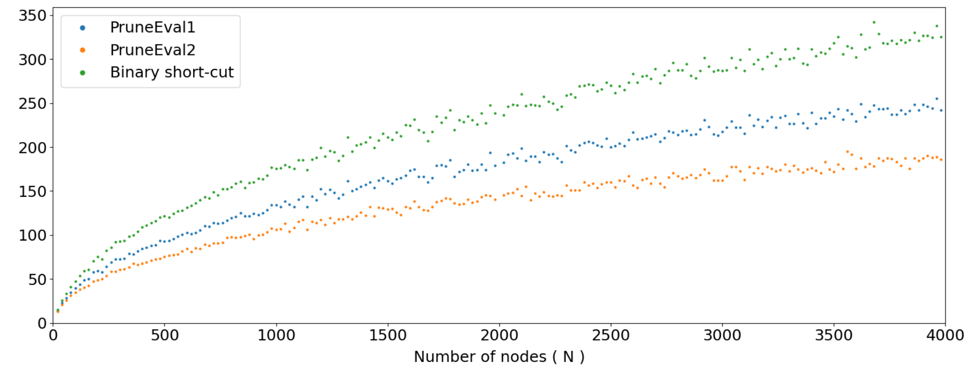


Figure 17: The number of not pruned nodes of Product fuzzy logic expression tree. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 17 shows how many nodes were not pruned when evaluating Product fuzzy logic formulae. Both algorithms have pruned efficiently. The ratio of the evaluated nodes is below 10% up to tree sizes of 4000 nodes, and just about 1.5% when closing to the large tree sizes of 100k nodes. The algorithm using the binary short-cut possibilities had to evaluate almost twice the number of nodes compared to our better pruning algorithm (PruneEval2 for Product fuzzy logic).



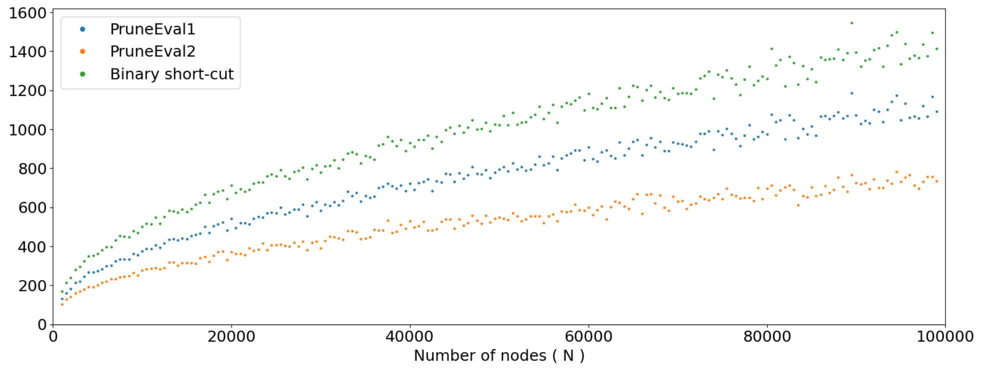


Figure 18: The number of not pruned nodes of Łukasiewicz fuzzy logic expression tree. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 18 shows the number of nodes that were touched when evaluating Łukasiewicz fuzzy logic formula tree. Both new pruning algorithms have been performed efficiently. The count of the not pruned nodes is around 5% with tree sizes of 4000 nodes, and just about 1% with the PruneEval2 Łukasiewicz version when closing to the large tree sizes of 100k nodes. We can compare the results of the new pruning algorithms with the performance of the algorithm using the well-known binary short-circuit methods during the evaluation. Similar to the previous results, our proposed new pruning algorithms performed much better, the algorithm PruneEval2 for Łukasiewicz fuzzy logic touching only 50% nodes compared to it.

All the above figures show the results with average values from many executions with different tree shapes and different value distributions. Now we will show the effect of the tree shape parameter on how effectively the pruning can be done. As we already mentioned, the shape of the formula tree has a significant effect on the efficiency of the pruning algorithms. In the following figures, we will show how efficient the pruning algorithms can be with several different tree shapes.

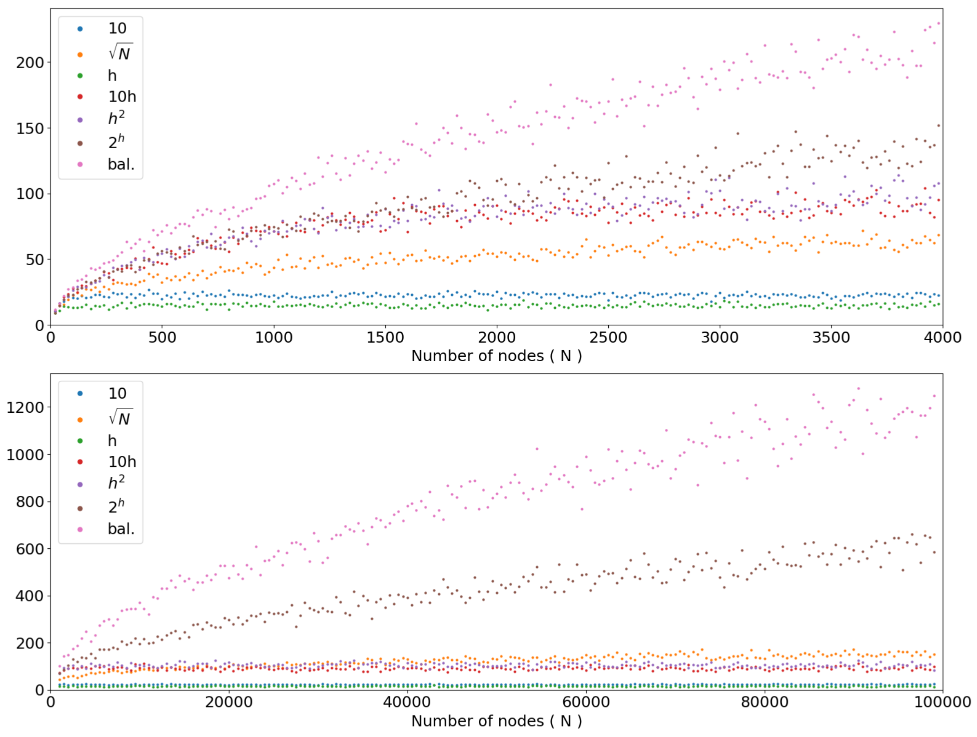


Figure 19: The number of nodes of Gödel formula trees not pruned by the Gödel version of the PruneEval2 algorithm, for various tree shapes (limit number of nodes at each h levels). See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 19 gives more detailed information on the number of the evaluated nodes for Gödel fuzzy logic expressions. As we see, the number of not pruned nodes is very low with the narrower trees. It is because leaf nodes have been found closer to the root node, and this led to an earlier cut. For wider trees, the ratio of the evaluated nodes is higher since the minimal depth is greater, where the algorithm can find the closest leaf nodes.

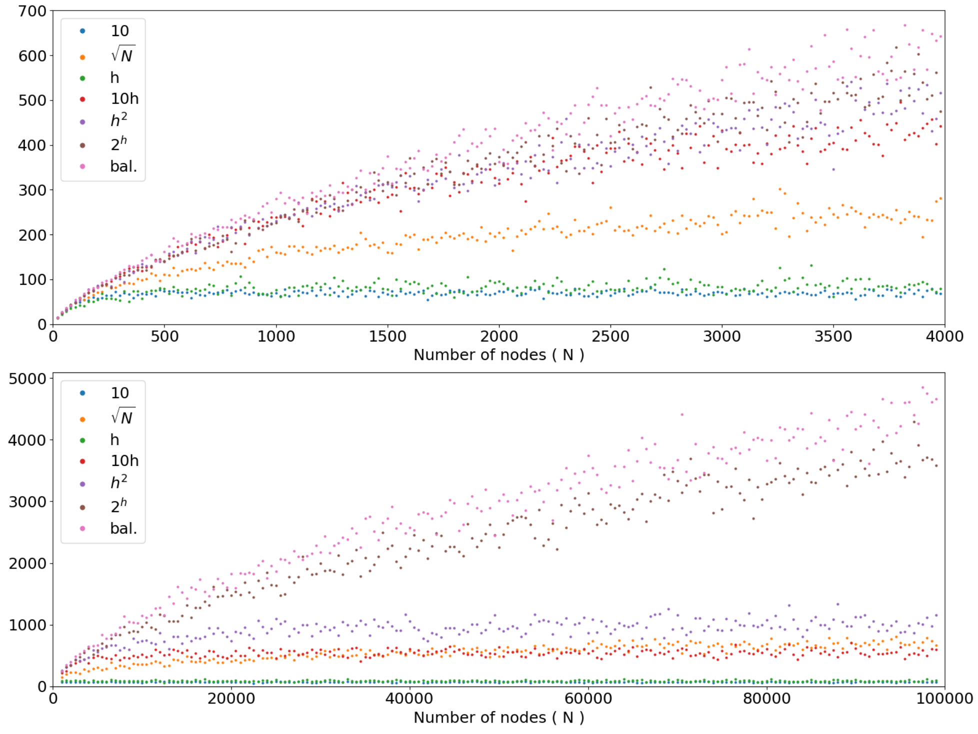


Figure 20: The number of nodes of the Product logic formula trees not pruned by the PruneEval2 algorithm for Product logic, for various tree shapes. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 20 shows the statistics of the evaluated nodes for Product logic expressions. For narrower formula trees for the same number of nodes, the number of the not pruned nodes is very low. In such trees there are leaf nodes closer to the root node, this can help the earlier cut. In the case of wider trees, the closest leaf nodes are at deeper levels, so the number of the evaluated nodes is higher.

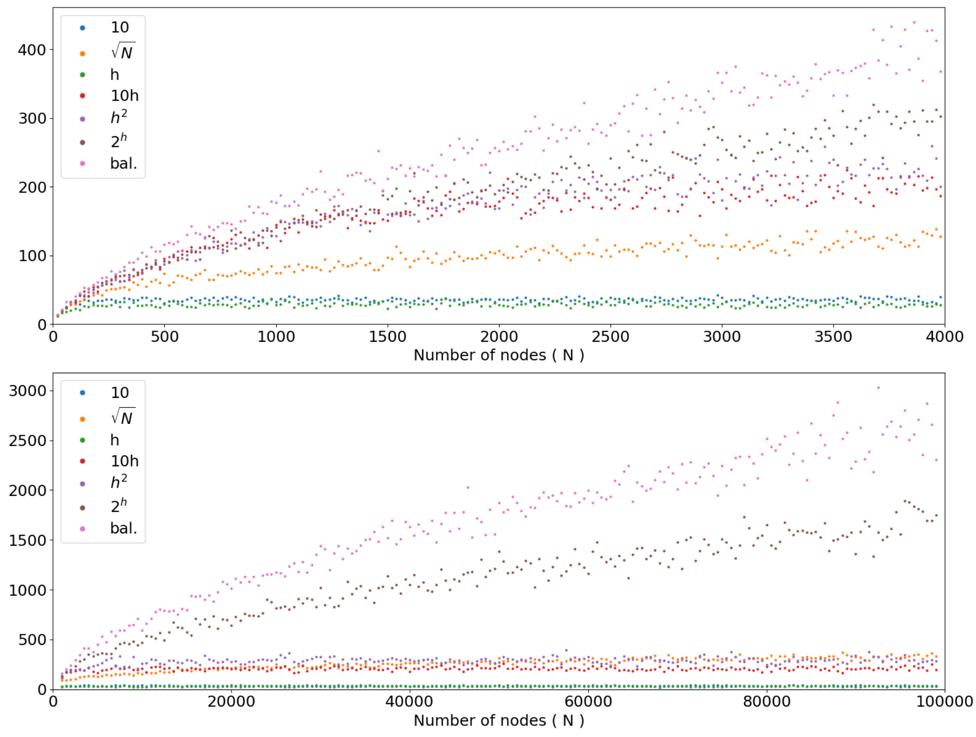


Figure 21: The number of nodes of Łukasiewicz formula trees not pruned by the PruneEval2 algorithm for various tree shapes. See the result for tree sizes with up to 4000 nodes above and up to 100000 nodes below.

Figure 21 gives more detailed information on the number of the evaluated nodes for Łukasiewicz logic expressions. As we see, the number of not pruned nodes is very low, similar to the other two logic types, for the narrower trees. This is because leaf nodes were found closer to the root node, and this led to an earlier short-cut. For wider trees, the number of evaluated nodes is larger because the minimum depth at which the algorithm finds the leaf nodes is larger.

7 Improvements for neural network classifiers

We propose simple additions that can be used for machine learning classifiers to possibly achieve better performance.

The proposed fuzzification of training data output class membership values can be used with standalone learners and with multiple (ensemble) learners as well for better results. This can be useful for datasets where the given training data has crisp (binary) class membership values although the "real" class membership values might be fuzzy. Such datasets can be e.g. images containing male and female faces, which might be misleading to have binary class membership values for each image, or weather data where e.g. the class "rain" could be fuzzy class as well. In addition to the possible performance improvement, this technique has an additional advantage as well. If the algorithm gives fuzzy class membership values, it has an additional level of information for the classification.

Another proposed addition is defining new variants for committee machine voting functions which, in some cases, might have better performance compared to the well-known voting functions. Having multiple possible voting functions we also provide meta voting functions which combine the output of the single voting functions.

These additions can be used separately or together as well.

We note that our experiment was done using neural network and convolutional neural network classifiers, however, these techniques might be used for every classifier, that can produce fuzzy output values, as well.

7.1 Fuzzification of neural network training data class membership crisp values

Neural network and convolutional neural network algorithms are powerful machine learning tools for solving classification or other problems. Their performance can certainly be influenced by the quality of the training data. One common issue is that training data usually have binary output values, even when the training samples may belong to more than one class at a certain fuzzy level [61, 19]. These data come usually labeled so that each sample has one or more labels, each of which means the crisp $True$ membership value in the class behind that label, and crisp $False$ membership value for the other classes in the same category. There can be cases where these crisp class membership values can be considered misleading, so the correction of these values can lead to reducing the confounding effect of them. Modification of training data is often useful for regularization. This can be done by e.g. making distortion, adding noise, using data augmentation [62], or adversarial training [63].

The proposed fuzzification technique might be applied to other classifying algorithms as well, in case they are able to give fuzzy membership values in their output. Research on other algorithms, in order to apply the fuzzification technique on them, can be future work, in the current research we conducted our measurements with neural network algorithms.

We define simple methods that can be used to modify the target output values given for the training patterns during the training process, to produce fuzzy target output values from the crisp (binary) values of the training data set. This class membership value fuzzification is done so that the knowledge gained during the learning process will be used to correct the inaccurate output class membership values of the training patterns. In the following, we will show and describe the proposed algorithm variants. The performance of these algorithm variations will be analyzed and shown in subchapter 8.2.

Three versions of the algorithm will be presented below. The first of them (Algorithm 7) is for single learners, the second version (Algorithm 8) is for multiple learners, the result of which can be used also with committee machine voting functions, the third variant (Algorithm 9) is a simple modification to handle the parameters of the fuzzification for multiple learners.

```
procedure FuzzyTraining(model,train_X,train_Y,FA,FB,FC):
  epoch = 0
  fuzzy_Y = train_Y
  while epoch<MAX_EPOCHS and CheckEarlyStopCondition()==False:
    model.fit(train_X, fuzzy_Y, epochs=1)
    out = model.predict(train_X)
    if epoch >= START_FUZZY:
      fuzzy_Y =  FA*fuzzy_Y + FB*out + FC*train_Y
    epoch = epoch + 1
```

Algorithm 7: Fuzzy Training

Algorithm 7 must be called with the training inputs and outputs, and the parameters for the fuzzification for the training of a learning model. The parameter FA is the weight for the momentum which means the importance of the actual (current) class membership values (the $fuzzy\_Y$ vector). This affects the change from original values towards the desired values giving the momentum for the actual knowledge. The parameter FB is the weight for the current knowledge (the out vector), which means the courage to change. The parameter FC is the weight used for the $train\_Y$ vector, which means the importance of the original target output data. The sum of the parameters FA, FB, and FC must be 1.0. In the above-presented algorithm, it is a simple condition to have some epochs before the first correction. This, of course, can be changed to an adaptive condition to achieve better performance, however, for our measurement, it is more important to know the number of correction operations. When the learning starts, the initial values in the $fuzzy\_Y$ vector are the same as given in the $train\_Y$ vector.

As it can be seen in Algorithm 7, the defined algorithm can work with individual learner algorithm. Our research had one such experiment which will be shown in chapter 8.2.

We give an extended variant of the algorithm as well, to enable us to use the combined knowledge of multiple (ensemble) learners. As we have discussed earlier, the usage of multiple learners of similar levels gives better results compared to the results of the individual learners. In this version of the algorithm, all the learners in the ensemble will modify the same $fuzzy\_Y$ corrected output values, so their combined opinion will have an effect on the subsequent training epochs.

```
procedure FuzzyTrainingEnsemble(models,train_X,train_Y,FA,FB,FC):
  epoch = 0
  fuzzy_Y = train_Y
  while epoch<MAX_EPOCHS and CheckEarlyStopCondition()==False:
    for model in models:
      model.fit(train_X, fuzzy_Y, epochs=1)
      out = model.predict(train_X)
      if epoch >= START_FUZZY:
        fuzzy_Y = FA*fuzzy_Y + FB*out + FC*train_Y
  epoch = epoch + 1
```

Algorithm 8: Fuzzy Training Ensemble

In the case of this new variant of the algorithm (Algorithm 8) the correction of the training data outputs will be better because the combined knowledge of the learners has a better performance compared to the individual results. The correction will also be faster because after every learning epoch of each individual learner a correction of the training data outputs will be done. In the case of multiple learners, parameter FA affects the change from original values towards the desired values and it affects the averaging effect on the outputs of multiple learners too. In future development, it might be useful to change the algorithm with an additional parameter to separately control these two effects. Of course, the condition when to start the correction must be also considered.

In this case, the number of times the correction statement will be run is the number of (epochs - START_FUZZY) multiplied by the number of the learners. This can be taken into account when setting the parameters for the training data output value fuzzification.

We provide a modification to Algorithm 8 with a simple normalization with respect to the number of learners.

Let M be the number of learners, FA, FB, and FC the weights for the training output class membership value fuzzification, as described for the algorithm. We can calculate the normalized FA', FB' and FC' weights as follows:

1) $FA' = \sqrt[M]{FA}$

2) $FB' = \frac{(1-FA')\cdot FB}{FB+FC}$

3) $FC' = 1 - (FA' + FB')$

The parameters FA', FB', FC' now correspond to the parameters FA, FB, FC so that the speed of convergence with M learners giving the same output will be the same as the speed of convergence would be using the parameters FA, FB, and FC with one learner.

Now we apply the above formulae to get the new version of this algorithm.

```
procedure FuzzyTrainingEnsemble2(models,train_X,train_Y,FA,FB,FC)
  :
  FA = power(FA, 1/len(models))
  FB = (1-FA)*FB/(FB+FC)
  FC = 1-(FA+FB)
  epoch = 0
  fuzzy_Y = train_Y
  while epoch<MAX_EPOCHS and CheckEarlyStopCondition()==False:
    for model in models:
      model.fit(train_X, fuzzy_Y, epochs=1)
      out = model.predict(train_X)
      if epoch >= START_FUZZY:
        fuzzy_Y = FA*fuzzy_Y + FB*out + FC*train_Y
    epoch = epoch + 1
```

Algorithm 9: Fuzzy Training Ensemble 2

In Algorithm 9 the normalization of the parameters has to be done only once, before the training loop. Certainly, it might be possible to adaptively change the parameters during the training process, the research on this can be conducted in the future.

Note that we have overwritten the original values of the parameters FA, FB, and FC. If this is not the desired behavior then these values can be preserved.

Since with a given number of learners and given (not changing) FA, FB, and FC parameters the difference between the Algorithm 8 and Algorithm 9 variants lies only on changing the parameters, we have not conducted any separate measurements on Algorithm 9.

7.2 New committee machine voting functions

We described some well-known committee machine voting functions in chapter 3.11. Their good performance motivated us to develop our new ones. We defined the following new committee machine voting functions which we will compare with some of the well-known voting functions. Some of them belong to the locally weighted average voting functions [40], others are meta voting functions.

- **Fuzzy average voting weighted by confidence (V2 in chapter 8.3):**

  Fuzzy average voting can be weighted by confidence [64]. Here we propose a simple function with getting confidence from the class membership values. This method obviously needs less performance compared to other more advanced methods. Class membership values closer to 0 or 1 will have a stronger weight, we transform the output of the individual learners before calculating the fuzzy average so that the values which are considered uncertain (not close to 0 or 1) will be less important by multiplying with a lower weight. Given the network output $o_i[j]$ for each i learner for each j class we calculate the combined result with the following formula with N learners:

  $$o[j] = \frac{1}{N}\sum_{i=1}^{N}((o_i[j] - 0.5) * (2 * o_i[j] - 1)^2 + 0.5)$$

  Then we get the winner class from this weighted average:

  $$l = argmax(o)$$

- **Fuzzy average voting weighted by 1-difference from the combined output (V3 in chapter 8.3):**

  Knowing the outputs of N learners, we can base another weighted average method based on the better performance of the fuzzy average voting compared to the individual learners. Starting with the calculation of the fuzzy average, individual predictions will be multiplied by a weight which is the difference from the ensemble prediction subtracted from 1. Let $o[j]$ for each j class be calculated as defined for the fuzzy voting in chapter 3.11. Then we calculate the new variant with N learners as follows:

  $$o'[j] = \frac{1}{N}\sum_{i=1}^{N}((o_i[j] - 0.5) * (1 - |o_i[j] - o[j]|) + 0.5)$$

We can find the winner class from the weighted average:

$$l = argmax(o')$$

- **Fuzzy average voting weighted by the reciprocal value of the number of failed training samples (V4 in 8.3):**

  Let f_i be the number of failed (misclassified) samples for each learner i on the training dataset if it is not equal to 0, otherwise, we use a value between 0 and 1, e.g. 0.5.

  The reciprocal value of f_i will be used as the weight for the learner i.

$$o[j] = \left(\frac{1}{N}\sum_{i=1}^{N}\frac{o_i[j]}{f_i}\right)\left(\frac{1}{N}\sum_{i=1}^{N}f_i\right)$$

  From this weighted average we get the winner class:

$$l = argmax(o)$$

- **Geometric mean (Nash voting with Nth root) (V7 in 8.3):**

  We create a variant of the Nash (product) vote function for using in meta voting function as well. Since with a higher number of voters (N) the product of many values from the interval $[0, 1]$ can be a very small number, much smaller than e.g. the fuzzy average so we take the Nth root of the product, getting the geometric mean of the output values.

  We note that the geometric mean will choose the same winner as the Nash (product) voting since the Nth root function is strictly monotonically increasing over the interval $[0, 1]$.

  For each j class evaluate the Nth root of the product sum of the predictions of all of the i individual learners:

$$o[j] = \sqrt[N]{\prod_{i=1}^{N} o_i[j]}$$

  Then find the class with the highest membership value:

$$l = argmax(o)$$

Meta-voting variants Fuzzy average or plurality vote by combining selected voting functions by calculating the fuzzy average or the plurality of votes on the classes of the results of the selected voting functions.

For analysis purposes, we define three meta voter variants.

- V8: Plurality voting from the results of V1, V2, V3, V4, V5, V6, V7
- V9: Plurality voting from the results of V1, V2, V3, V4, V7
- V10: Fuzzy average voting from the results of V1, V2, V3, V4, V7

For the above three meta voting functions, we calculate the results of the required voting functions first, then we combine them as it was described earlier for the voting functions calculated from the results of the individual learners.

We note that any data used to calculate the weights certainly can only be part of the training data or result of the learning process, without any knowledge about test data or performance on test data.

We note also that plurality vote and Borda vote functions do not give fuzzy class membership values, so they cannot be combined well by fuzzy average with the fuzzy results of other voters. For the performance evaluation, we will use the three meta voter functions described above (V8, V9, V10) for better understanding and comparison.

8 Performance evaluation of fuzzification and voting functions

8.1 Performance evaluation framework

The experiments ran on personal computers equipped with NVIDIA and AMD GPUs using Tensorflow from Python programs. Our simple framework was based on a file interface that enables us to run the machine learning on multiple machines, and then later collect and process the output files generated by the learners. For the research, one multi-layer perceptron model and two convolutional neural network learning algorithms with different strengths have been chosen as the basis of the modifications [65, 58]. The problem set given to the learning algorithms was the well-known MNIST database of handwritten digits [56] (see subchapter 3.14). We plan to perform research on other problem sets as well. The results may vary given the stochastic nature of the algorithms, so thousands of experiments with different parameters were performed, and average results were analyzed. For the analyses, we first measured the performance of the individual learners on the test dataset with different parameters for fuzzification. Since multiple learners have proven to be more successful when we combine their results through voting, we might expect better results producing the fuzzified class membership together as well. In this experiment ensemble learners in different group sizes were run. We will show the results of this research in the next subchapters. In these experiments, we have measured the standalone test results of the learners, as well as the results of the fuzzy average voting of the groups. When we talk about committee machine voting, we can choose from many voting functions, e.g. fuzzy averaging, plurality(or majority) voting, etc. In our research, we have compared the results of some of the most well-known voting functions with our newly defined ones.

For the analyses, we used the Python Numpy and Pandas frameworks.

The algorithms run with different epoch counts to see the behavior of our proposed algorithm variations not only with the statistically best settings.

In the following two subchapters, we will show the performance of the proposed fuzzification of training data binary class membership values and the proposed voting functions.

For the evaluation, we run a total of about one million learning sessions with one MLP and two convolutional neural network algorithms, modified according to our proposed methods.

The first algorithm variant used a very simple multi-layer perceptron (MLP) with 1024 and 128 neurons in the hidden layers, the activation function of which was the relu (rectified linear unit), the activation function selected for the output was the softmax.

The second algorithm was built from the algorithm introduced in [65]. It is a simple convolutional neural network algorithm, the network architecture of which will be shown below.

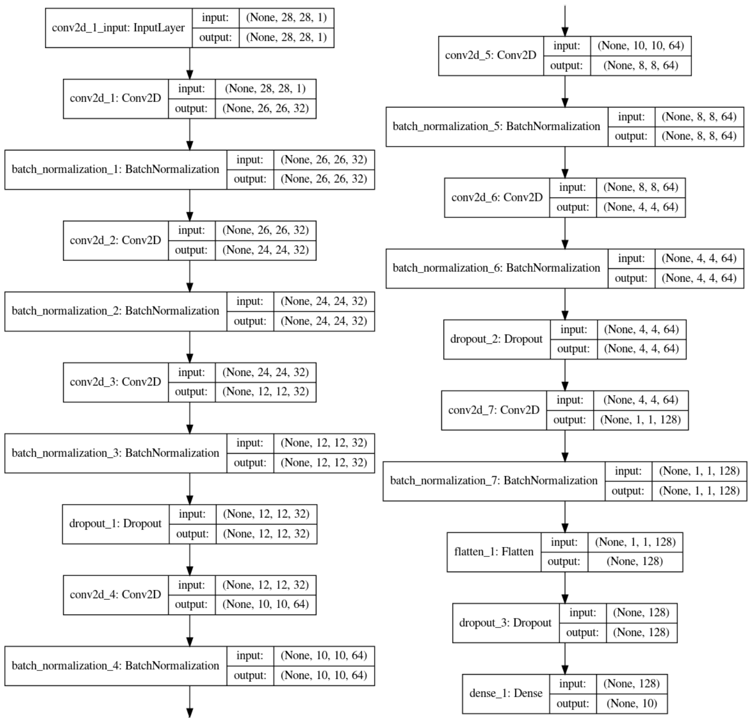
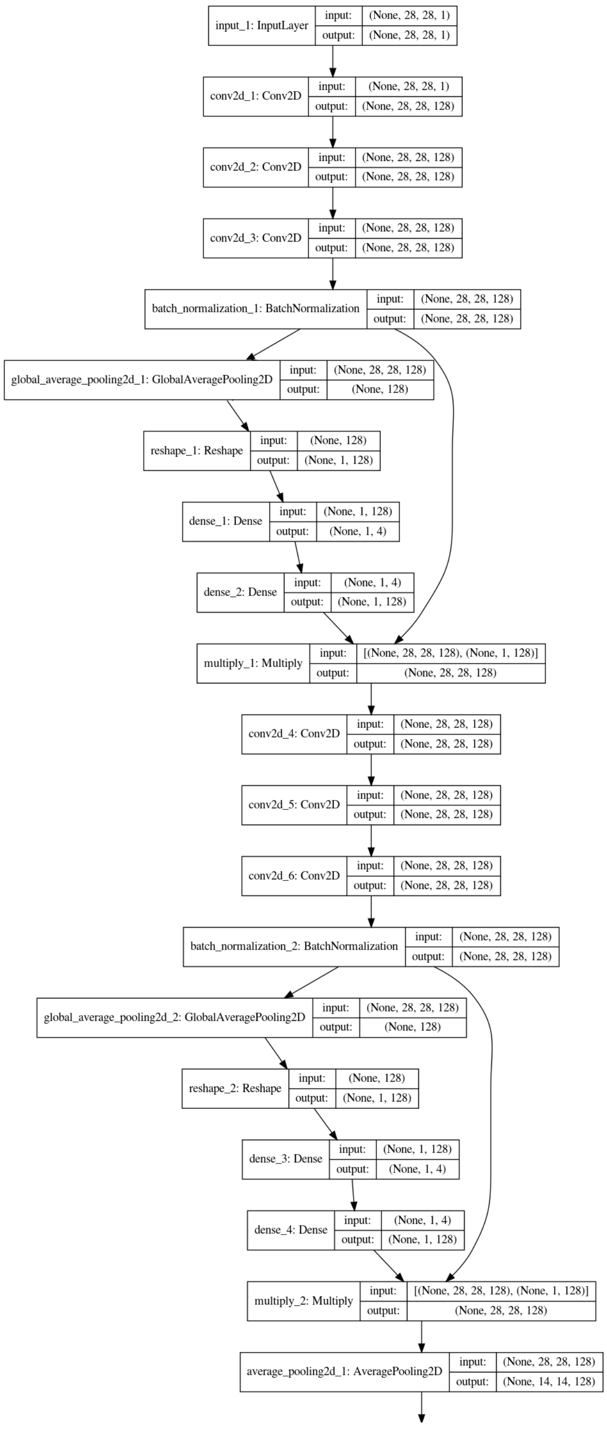

conv2d_1_input: InputLayer | input: (None, 28, 28, 1) | output: (None, 28, 28, 1)
conv2d_1: Conv2D | input: (None, 28, 28, 1) | output: (None, 26, 26, 32)
batch_normalization_1: BatchNormalization | input: (None, 26, 26, 32) | output: (None, 26, 26, 32)
conv2d_2: Conv2D | input: (None, 26, 26, 32) | output: (None, 24, 24, 32)
batch_normalization_2: BatchNormalization | input: (None, 24, 24, 32) | output: (None, 24, 24, 32)
conv2d_3: Conv2D | input: (None, 24, 24, 32) | output: (None, 12, 12, 32)
batch_normalization_3: BatchNormalization | input: (None, 12, 12, 32) | output: (None, 12, 12, 32)
dropout_1: Dropout | input: (None, 12, 12, 32) | output: (None, 12, 12, 32)
conv2d_4: Conv2D | input: (None, 12, 12, 32) | output: (None, 10, 10, 64)
batch_normalization_4: BatchNormalization | input: (None, 10, 10, 64) | output: (None, 10, 10, 64)
conv2d_5: Conv2D | input: (None, 10, 10, 64) | output: (None, 8, 8, 64)
batch_normalization_5: BatchNormalization | input: (None, 8, 8, 64) | output: (None, 8, 8, 64)
conv2d_6: Conv2D | input: (None, 8, 8, 64) | output: (None, 4, 4, 64)
batch_normalization_6: BatchNormalization | input: (None, 4, 4, 64) | output: (None, 4, 4, 64)
dropout_2: Dropout | input: (None, 4, 4, 64) | output: (None, 4, 4, 64)
conv2d_7: Conv2D | input: (None, 4, 4, 64) | output: (None, 1, 1, 128)
batch_normalization_7: BatchNormalization | input: (None, 1, 1, 128) | output: (None, 1, 1, 128)
flatten_1: Flatten | input: (None, 1, 1, 128) | output: (None, 128)
dropout_3: Dropout | input: (None, 128) | output: (None, 128)
dense_1: Dense | input: (None, 128) | output: (None, 10)

Figure 22: Network architecture for the first convolutional neural network variant created from the algorithm [65].

Figure 22 shows the model which was built in the Keras/Tensorflow framework. The figure was generated from the model using the framework. It shows the layers used in the model and how they are connected.

Similarly, we also will show the network architecture used in the third algorithm variant as well. This variant was based on the algorithm [58] which uses the Squeeze-and-Excitation Network method.

Figure 23 shows the neural network architecture which we used for our third neural network program. This program was based on the algorithm [58] using Squeeze-and-Excitation architecture. We have added the proposed fuzzification of binary class membership values of training data and also the proposed voting functions.


input_1: InputLayer | input: (None, 28, 28, 1) | output: (None, 28, 28, 1)
conv2d_1: Conv2D | input: (None, 28, 28, 1) | output: (None, 28, 28, 128)
conv2d_2: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
conv2d_3: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
batch_normalization_1: BatchNormalization | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
global_average_pooling2d_1: GlobalAveragePooling2D | input: (None, 28, 28, 128) | output: (None, 128)
reshape_1: Reshape | input: (None, 128) | output: (None, 1, 128)
dense_1: Dense | input: (None, 1, 128) | output: (None, 1, 4)
dense_2: Dense | input: (None, 1, 4) | output: (None, 1, 128)
multiply_1: Multiply | input: [(None, 28, 28, 128), (None, 1, 128)] | output: (None, 28, 28, 128)
conv2d_4: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
conv2d_5: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
conv2d_6: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
batch_normalization_2: BatchNormalization | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
global_average_pooling2d_2: GlobalAveragePooling2D | input: (None, 28, 28, 128) | output: (None, 128)
reshape_2: Reshape | input: (None, 128) | output: (None, 1, 128)
dense_3: Dense | input: (None, 1, 128) | output: (None, 1, 4)
dense_4: Dense | input: (None, 1, 4) | output: (None, 1, 128)
multiply_2: Multiply | input: [(None, 28, 28, 128), (None, 1, 128)] | output: (None, 28, 28, 128)
average_pooling2d_1: AveragePooling2D | input: (None, 28, 28, 128) | output: (None, 14, 14, 128)

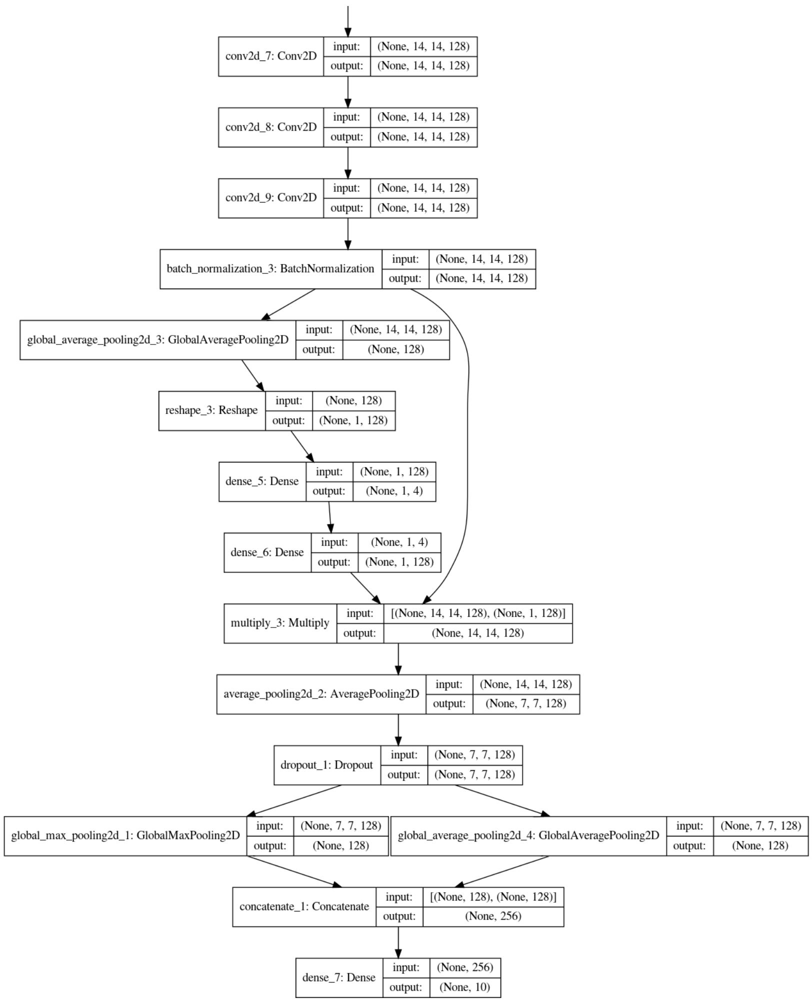

conv2d_7: Conv2D | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
conv2d_8: Conv2D | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
conv2d_9: Conv2D | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
batch_normalization_3: BatchNormalization | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
global_average_pooling2d_3: GlobalAveragePooling2D | input: (None, 14, 14, 128) | output: (None, 128)
reshape_3: Reshape | input: (None, 128) | output: (None, 1, 128)
dense_5: Dense | input: (None, 1, 128) | output: (None, 1, 4)
dense_6: Dense | input: (None, 1, 4) | output: (None, 1, 128)
multiply_3: Multiply | input: [(None, 14, 14, 128), (None, 1, 128)] | output: (None, 14, 14, 128)
average_pooling2d_2: AveragePooling2D | input: (None, 14, 14, 128) | output: (None, 7, 7, 128)
dropout_1: Dropout | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
global_max_pooling2d_1: GlobalMaxPooling2D | input: (None, 7, 7, 128) | output: (None, 128)
global_average_pooling2d_4: GlobalAveragePooling2D | input: (None, 7, 7, 128) | output: (None, 128)
concatenate_1: Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
dense_7: Dense | input: (None, 256) | output: (None, 10)

Figure 23: Network architecture for the second convolutional neural network variant for our performance evaluation based on the algorithm [58].

8.2 Performance evaluation of training data class membership value fuzzification

We have executed several experiments with three algorithms with different strengths. The algorithm variations were executed with different parameters, e.g. the number of epochs to run, the number of instances in the ensembles, and the parameters for the fuzzification of binary class membership values of training data, including parameters that keep the original class membership values. We note that we have executed thousands of learning sessions without fuzzification in order to have more reliable results. For the fuzzification experiments, we made the $FUZZY\_START = 1$ setting, i.e. with each learner, the change of the class membership values of the training data was executed after each epoch, except after the first one.

8.2.1 Fuzzification experiment 1

The first experiment was executed using a very simple multi-layer perceptron architecture with 784, 1024, 128, 10 neurons in the successive layers. The activation function of the hidden layers was the relu, the output activation function was the softmax. This algorithm can be considered a weak learner, compared to the current state-of-the-art algorithms. Thousands of learners learned 20 epochs in ensembles of 10 learners. The ensembles had different parameters for fuzzification, including parameters that keep the original class membership values ($FA = 0, FB = 0, FC = 1$). Input data distortions were applied during the training.

We will first show the average results as a function of the $FC/(FB + FC)$ ratio.

Figure 24 shows the average accuracy of the individual learners with different parameters used for the fuzzification. The ratio $FC/(FB + FC)$ of the fuzzifying parameters of our algorithms has the meaning of how important the original binary class membership values provided in the training data are. If the ratio is 1.0, then no fuzzification occurs. As can be seen, the accuracy achieved was better when the algorithm was used with fuzzification.

We will also show the performance using the fuzzy average voting function when using multiple learners.

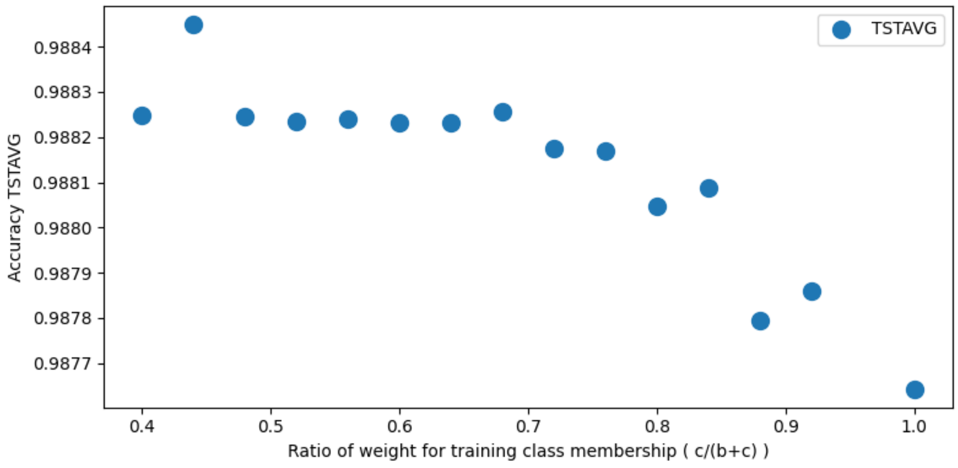


Figure 24: The average accuracy results of our MLP based algorithm on test data using different parameters for the fuzzification of the training data class membership values.

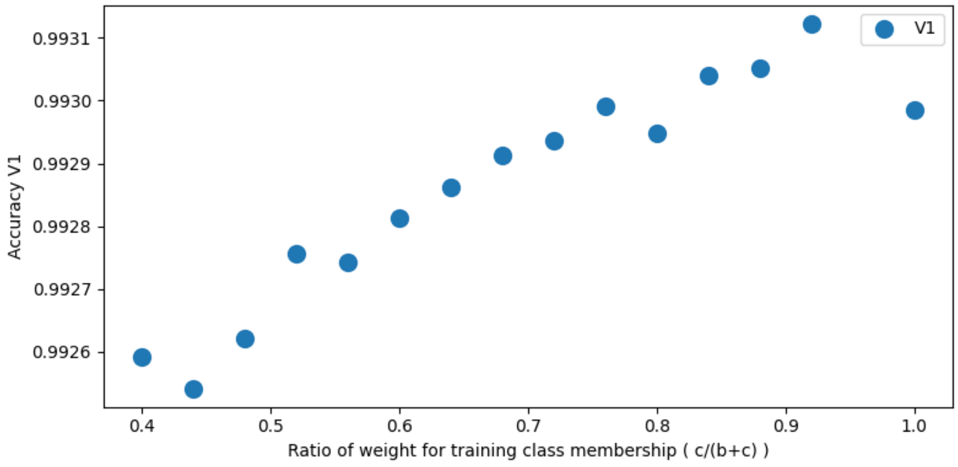


Figure 25: The performance results of our algorithms with V1 fuzzy average voting function by $5-20$ voters on test data using different parameters for the fuzzification of the training data class membership values.

Figure 25 shows the results of the V1 fuzzy average voting of the same experiment. As we can see, the results using the fuzzy average voting are different compared to the individual results, fuzzification could help to achieve better accuracy of the algorithm on the training dataset only for a range of the fuzzification parameters.

Since the fuzzification is controlled with multiple parameters, we also show 3D diagrams to better understand the results for different parameters. Since the sum of the parameters FA, FB, and FC must be 1.0 (see subchapter 7.1) we can choose two of these parameters for the X and Y axes of the diagram, and the Z-axis can show the average accuracy values. We have chosen the FA and FB parameters for that, the FC parameter for every measurement is $1 - (FA + FB)$.

First, we will show the individual accuracies of the learners with this method.

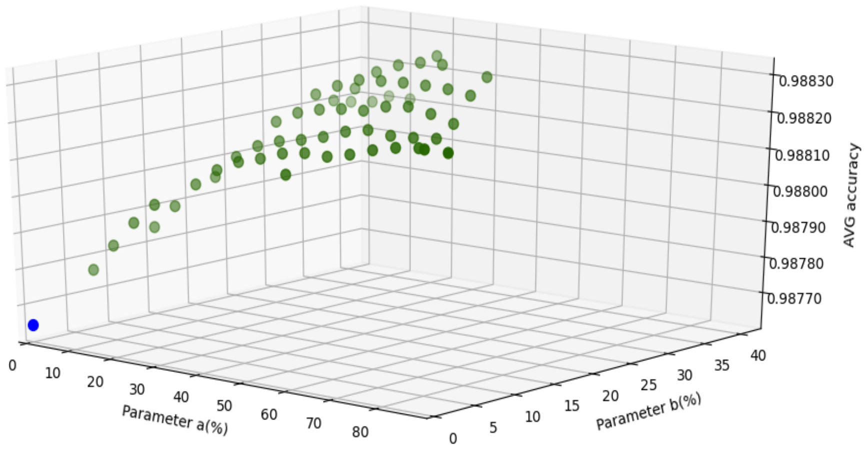


Figure 26: The average performance results of our algorithms on test data, using different parameters for the fuzzification of the training data class membership values.

Figure 26 shows the average individual accuracy of the learners on test data for the FA and FB parameters (parameter values shown in %). The value with $FA = 0, FB = 0$ coordinates shows the average result without fuzzification. We can see that with values of parameter FB around 40 we had better accuracy, especially when the value of parameter FA was close to 20. This means that the stronger fuzzification with a stronger momentum factor resulted in better accuracy. The difference was significant.

Below we also show a 3D diagram to see the performance of the fuzzy average voting (V1) results from the same experiment.

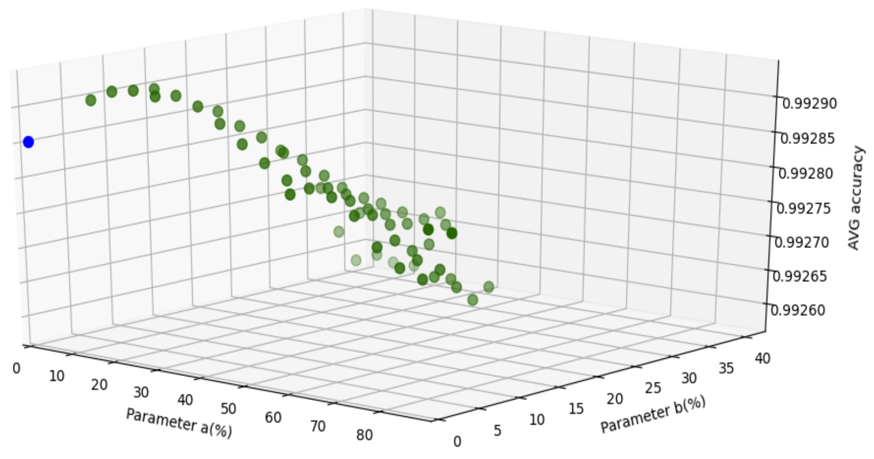


Figure 27: The average performance results of our algorithms' fuzzy average voting on test data using different parameters for the fuzzification of the training data class membership values.

Figure 27 shows the average accuracy of the algorithm on test data for the FA and FB parameters (parameter values shown in %). The value with $FA = 0, FB = 0$ coordinates shows the average result without fuzzification. We can see that only the ensemble learning executions with fuzzification performed with lower values of the FA and FB parameters were able to achieve better accuracy using committee machine fuzzy average voting, compared to the learning done without fuzzification.

It is very interesting to see on the previous two figures, that the learning sessions with the best individual results had about the lowest accuracy using them as committee machine.

8.2.2 Fuzzification experiment 2

The second experiment was performed using the modified variant of a convolutional neural network [65]. Thousands of learners learned a different number of epochs and with different parameters for fuzzification, including parameters that keep the original class membership values ($FA = 0, FB = 0, FC = 1$).

We will first show the average results as a function of the $FC/(FB + FC)$ ratio.

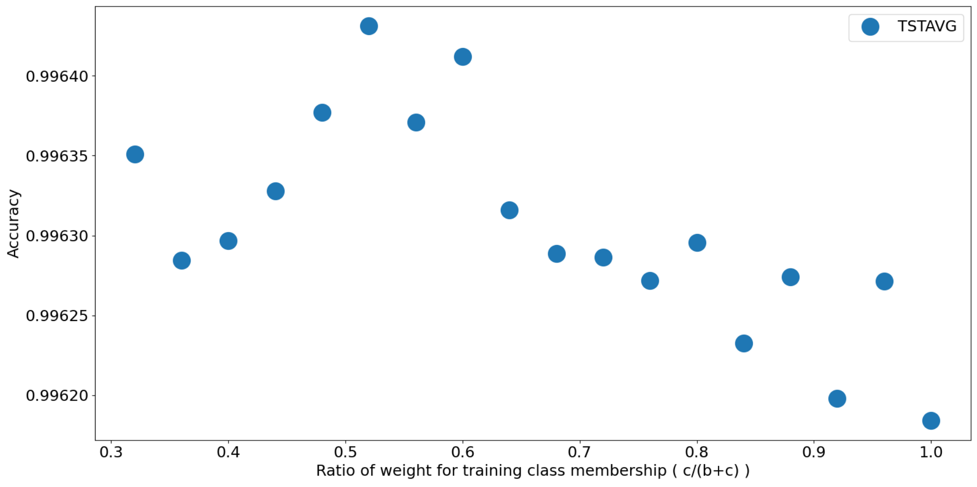


Figure 28: The average accuracy results of our algorithm on test data using different parameters for the fuzzification of the training data class membership values.

Figure 28 shows the learners' average prediction accuracy with different parameters used for the fuzzification. The ratio $FC/(FB+FC)$ of the fuzzifying parameters of our algorithms shows the importance of the original binary class membership values given in the training data. If the ratio is 1.0 , then no fuzzification will happen. We can see that the performance using fuzzification was better.

In the following, we also show the performance using the fuzzy average voting function when using multiple learners.

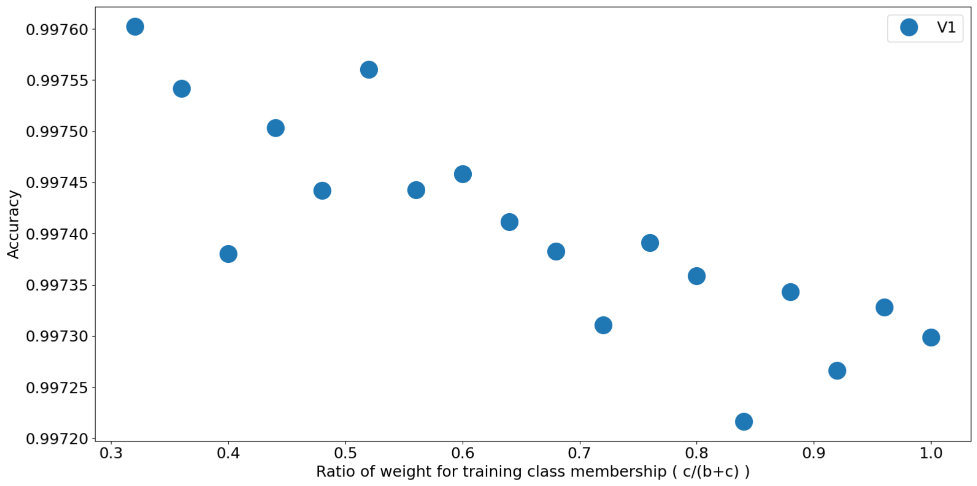


Figure 29: The average performance results of our algorithms V1 fuzzy average voting function by $5-20$ voters on test data using different parameters for the fuzzification of the training data class membership values.

Figure 29 presents the results of the V1 fuzzy average voting on the same experiment. As we can see, the results using the fuzzy average voting are similar, the fuzzification helps to achieve better performance, i.e. higher accuracy of the prediction on the training dataset. As the ratio of $FC/(FB+FC)$ increases, i.e., the possibility of fuzzification decreases, so the accuracy achieved tends to decrease as well. We note, that although the presented results are mean values of several measurements, the stochastic behavior of the algorithms can result in fluctuations in performance, e.g. the two average values with lower accuracy compared to the non-fuzzified version can be the effect of that.

We also show a 3D diagram to better understand the results using different parameters for fuzzification. Since the sum of the parameters FA, FB, and FC must be 1.0 (see subchapter 7.1) we have chosen the FA and FB parameters for the X and Y axes, the FC parameter for every measurement is $1-(FA+FB)$.

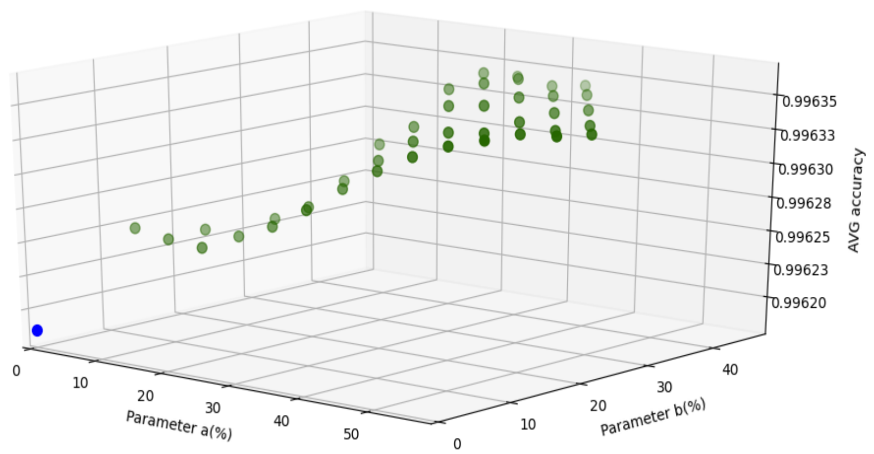


Figure 30: The average accuracy results of our algorithms on test data using different parameters for the fuzzification of the training data class membership values.

Figure 30 shows the average accuracy of the algorithm on test data for the various FA and FB parameters. The result with coordinates $FA = 0, FB = 0$ shows the average result without fuzzification. The stronger fuzzification resulted in better accuracy, especially when the value of the FA parameter setting the factor of the current knowledge (momentum) was between 20% and 30%. The difference between the individual accuracy of the algorithms with and without fuzzification was significant.

As in the previous experiment, there was an interesting difference between the behavior of the individual results and the results using committee machines, we present below the fuzzy average voting results for this experience as well.

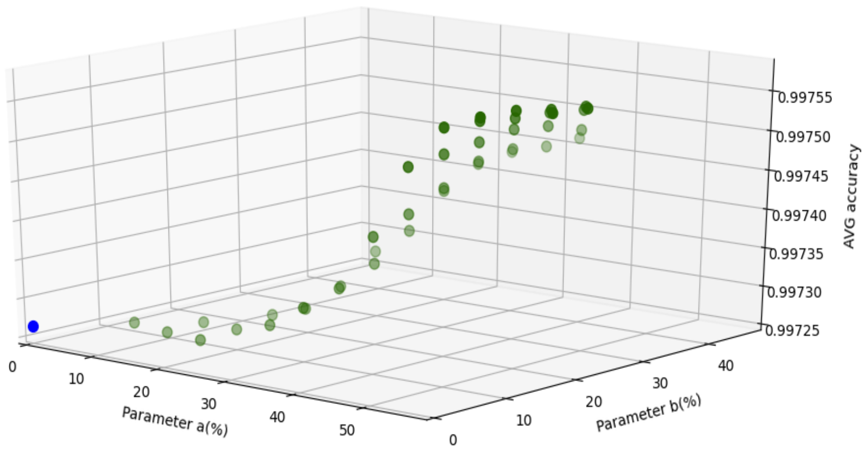


Figure 31: The average performance results of our algorithms used as committee machine fuzzy average voting with up to 20 voters on test data using different parameters for the fuzzification of the training data class membership values.

Figure 31 shows the average accuracy achieved on test data by up to 20 voters using the fuzzy average voting with different FA and FB parameters. The value with $FA = 0, FB = 0$ coordinates shows the average result without fuzzification. With values of parameter FB around 40%, we had better accuracy, especially when the value of parameter FA was close to 30%.

These results are different from the previous experiment. Now we can see that stronger fuzzification led to better accuracy with using committee machine fuzzy average voting, as well. We examined the possible reasons for this. We found that in this experiment the results of the voting functions had negative correlation with the accuracy on the training data. We also found that with that algorithm the V4 voting function had worse result and weaker correlation to other voting functions. This both can mean that we had greater overfitting in that experiment, and stronger fuzzification had a benefit to lower the overfitting.

8.2.3 Fuzzification experiment 3

In the third experiment, we executed the learning sessions using a modified algorithm of [58], using the parameters for the fuzzification. The number of epochs we had executed our algorithm was set from 15 to 20.

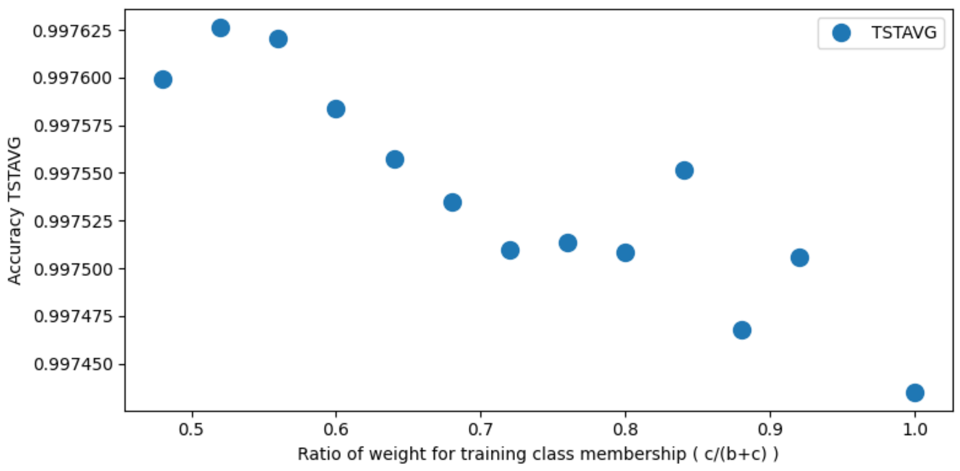


Figure 32: The individual accuracy results of the algorithm on test data using different parameters for the fuzzification of the training data class membership values.

Figure 32 shows the average accuracy of the individual learners with different parameters used for the fuzzification. As we already described for Figure 24, the ratio FC/(FB+FC) tells the importance of the original class membership values of the training data, fuzzification can be done only if the ratio is below 1.0. As we can see, the accuracy can be better with modest fuzzification. We also note that if the ratio of FC/(FB+FC) decreases to about 0.5 or below, then the accuracy decreases as well. This can be the effect of too much freedom of the algorithm to change the class membership values.

For this experiment, too, we have measured the performance using the well-known fuzzy average voting function (V1) when using multiple (5-20) learners.

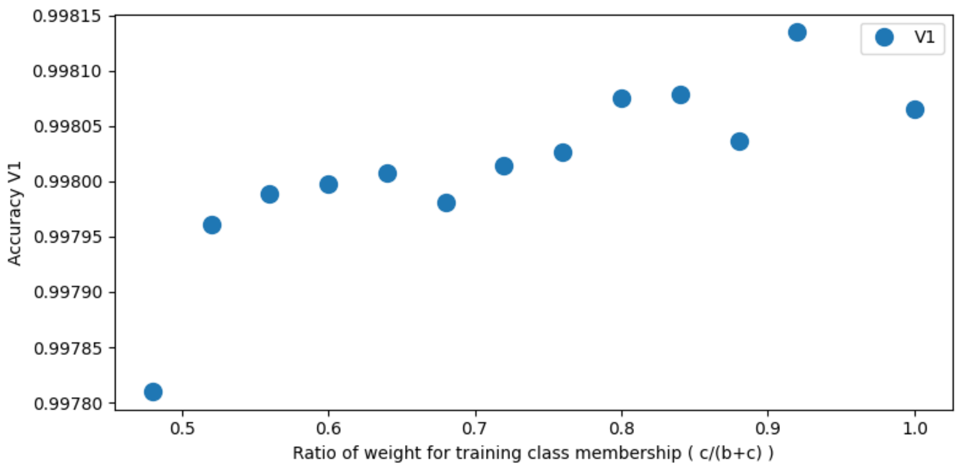


Figure 33: The performance results of our algorithms V1 fuzzy average voting function by $5-20$ voters on average on test data using different parameters for the fuzzification of the training data class membership values.

Figure 33 shows the results of the V1 fuzzy mean vote in this experiment. The results are different in this case. The accuracy averages using fuzzified training data class membership values were lower for most parameters compared to the accuracy using only the original training data. However, there is a promising range which we can look at from another perspective, as well.

Below we show the average accuracy of the learners for different FA, FB, and FC parameters of the fuzzification algorithm on a 3D figure.

First, the performance of individual predictions will be shown.

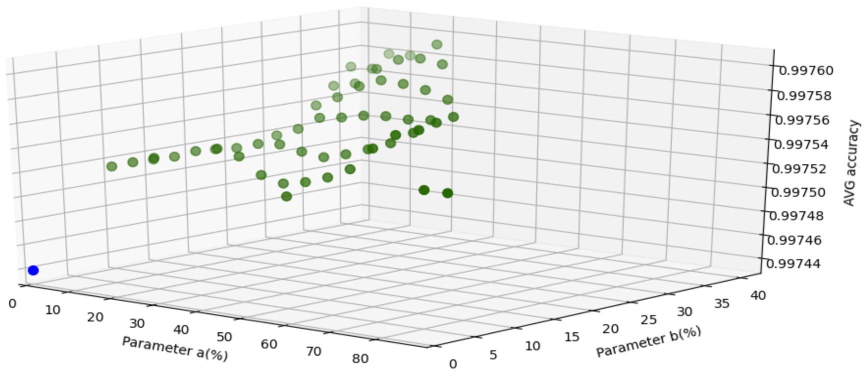


Figure 34: The results of our algorithms average accuracy on test data using different parameters for the fuzzification of the training data class membership values.

Figure 34 shows the results of thousands of learning sessions that were executed with different FA, FB, and FC parameters. The point with $FA = 0, FB = 0$ coordinates shows the average result when class membership values of training data were not corrected. We can see that the results were higher with lower FA and FB parameter values. For such parameters the FC parameter is higher, so only minor corrections on the training data class membership values can be made.

Similar to the previous experiments the performance of fuzzification was significant with this strong algorithm, as well. Although the algorithm without performing fuzzification has a good performance, adding our method to correct the class membership of the training data improved the accuracy of the individual learners. The stronger fuzzification, i.e. larger value for the factor of the actual output of the learner, resulted in higher accuracy.

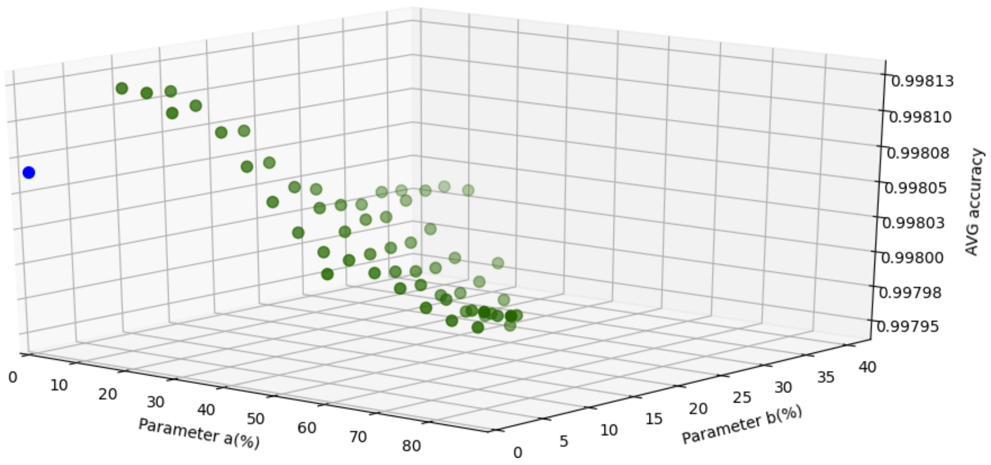


Figure 35: The results of our algorithms fuzzy average (V1) voting accuracy on test data using different parameters for the fuzzification of the training data class membership values.

Figure 35 shows the committee machine fuzzy average voting performance (with $5-20$ voters) of thousands of learning sessions which were executed with different FA, FB, and FC parameters. The point with $FA = 0, FB = 0$ coordinates shows the average result when class membership values of training data were not corrected. We can see that the results were higher with lower FA and FB parameter values. For such parameters the FC parameter is higher, so only minor corrections on the training data class membership values can be made in these cases.

This is a strongly different behavior compared to the performance of fuzzification with the previous (weaker) algorithm variant.

For a range of parameter values, where parameter FA and FB are not zero but both have low values, the accuracy was better using the proposed fuzzification. That means that the fuzzification at a lower rate had an improvement even for this strong algorithm.

8.2.4 Fuzzification experiment 4

Finally, we examined the behavior of the fuzzification algorithm, based on the same squeeze-and-excite convolutional neural network, comparing the results with different number of parallel learners, changing the class membership values of training data together during the training in each epoch after $START\_FUZZY = 1$ epochs already done, using the fuzzification algorithm 8.

This evaluation has run the same algorithm as in the previous Fuzzification experiment 3.

In this experiment about 500 training sessions were executed, there were a total of about 5000 learner instances. We run the convolutional neural network algorithm with our fuzzification algorithm ensemble variant (Algorithm 8) in sessions with $1, 2, 5, 7, 12$ and 20 parallel learners. We also include the results of learning sessions performed with no fuzzification (labeled with $m = 0$) for comparison.

As we can see from Table 1, the average accuracy of the predictions of the learners was better for a larger number of parallel learners. This is in line with our expectations since the accumulated knowledge of the ensemble is usually better than the knowledge of the individual learners. In the case of one learner, the improvement was smaller, we got better and better results with the increasing number of learners.

epoch	m=0	m=1	m=2	m=5	m=8	m=12	m=20
1	0.976588	0.976200	0.975500	0.975685	0.975005	0.974602	0.974305
2	0.981150	0.982004	0.982854	0.983807	0.984461	0.985054	0.986008
3	0.984246	0.984741	0.985561	0.986446	0.986917	0.987526	0.988113
4	0.987010	0.987690	0.987921	0.988603	0.988896	0.989314	0.989682
5	0.989467	0.989684	0.989809	0.990218	0.990281	0.990529	0.990716
6	0.990244	0.990641	0.990750	0.991144	0.991263	0.991473	0.991735
7	0.991383	0.991599	0.991652	0.991996	0.992034	0.992154	0.992298
8	0.990086	0.990347	0.991007	0.991630	0.991954	0.992372	0.992846
9	0.992601	0.992734	0.992723	0.992967	0.992961	0.993020	0.993131
10	0.992611	0.992810	0.992973	0.993259	0.993345	0.993484	0.993690
11	0.993238	0.993417	0.993469	0.993683	0.993744	0.993897	0.994013
12	0.993132	0.993403	0.993498	0.993767	0.993843	0.994006	0.994186
13	0.993731	0.993921	0.993978	0.994160	0.994233	0.994337	0.994471
14	0.996517	0.996531	0.996498	0.996613	0.996584	0.996598	0.996630
15	0.996719	0.996731	0.996711	0.996798	0.996773	0.996778	0.996804
16	0.996813	0.996841	0.996837	0.996921	0.996912	0.996924	0.996955
17	0.997011	0.997032	0.997036	0.997100	0.997095	0.997106	0.997132
18	0.997157	0.997171	0.997181	0.997229	0.997231	0.997239	0.997261
19	0.997254	0.997269	0.997293	0.997335	0.997349	0.997362	0.997387
20	0.997550	0.997560	0.997579	0.997591	0.997604	0.997607	0.997619

Table 1: Average accuracy of individual predictions on the test samples for ensemble learning voting experiment 4

8.3 Performance of voting functions

For the evaluation, we have included the well-known voting schemes and our new variants as well.

We have the following voters implemented (see subchapters 3.11, and 7.2):

- V1: fuzzy voting, i.e. averaging
- V2: fuzzy variant - the average of individual predictions weighted by a confidence estimation of the class membership values
- V3: fuzzy variant - the average of individual predictions weighted by 1-difference from V1 results, predictions will be multiplied by a weight which is the difference from the ensemble prediction subtracted from the value 1.0
- V4: fuzzy variant - the average of individual predictions weighted by 1/training failures
- V5: plurality voting
- V6: Borda voting
- V7: geometric mean voting (instead of product voting)
- V8: meta-voter: plurality vote by using all the above voting functions
- V9: meta-voter: plurality meta vote of voters without the plurality and Borda voting (V1-V4, V7)
- V10: meta voter: fuzzy average meta vote of voters without the plurality and Borda voting (V1-V4, V7)

We note that variations of the plurality vote also can be applied [66], however, plurality and Borda votes, according to our measurements, are not among the best performing voting functions, we included them for reference and comparison purposes.

8.3.1 Voting experiment 1 with our algorithm based on simple multi-layer perceptron (MLP)

The first experiment ran 1000 training sessions based on a very simple neural network algorithm, an MLP architecture with 784, 1024, 128, 10 neurons in the successive layers. It was executed with different epoch counts (16,18,20) to eliminate the effect of a possibly statistically optimized epoch count for a specific dataset.

voting function	**MIN**	**AVG**	**MAX**
min(accuracy)	0.981700	0.986263	0.990600
avg(accuracy)	0.983600	0.988148	0.990900
max(accuracy)	0.983800	0.989761	0.991900
V1–fuzzy average	0.987600	0.992755	0.994800
V2–weighted by confidence	0.987600	0.992769	0.994700
V3–weighted by diff from V1	0.987600	0.992752	0.994800
V4–weighted by 1/training fails	0.987600	0.992761	0.994700
V5–plurality voting	0.981100	0.992449	0.994800
V6–Borda voting	0.982000	0.992537	0.994600
V7–geometric mean voting	0.987500	**0.992790**	0.994800
V8–meta–plurality (V1-V7)	0.987600	0.992764	0.994800
V9–meta–plurality (V1-V4, V7)	0.987600	0.992765	0.994800
V10–meta–fuzzy avg (V1-V4, V7)	0.987700	0.992776	0.994800

Table 2: Minimum, maximum, and average accuracy for ensemble learning voting experiment 1

Table 2 shows the results where $5-20$ voters cast their votes in each turn which were then combined using the above-defined voter functions. The best individual result was 81 misclassification on 10000 test samples, the worst individual result was 183 failed samples and on average they performed only 118.52 fails as individual learners. The well-known fuzzy voting performed 72.45 misses on average. There were no big differences among the voting functions, the best average result (72.1 fails on average) was achieved by the geometric mean (V7) voting function.

Also, we can check whether the difference between the voting functions depends on the number of voters. In the next figure, we can check that for some of the best performing voting functions.

It is also interesting to see the performances for different sizes of the committees. It is obvious to expect better results with a larger number of learners. Below we show the accuracy of some voting functions, which are among the best performers.

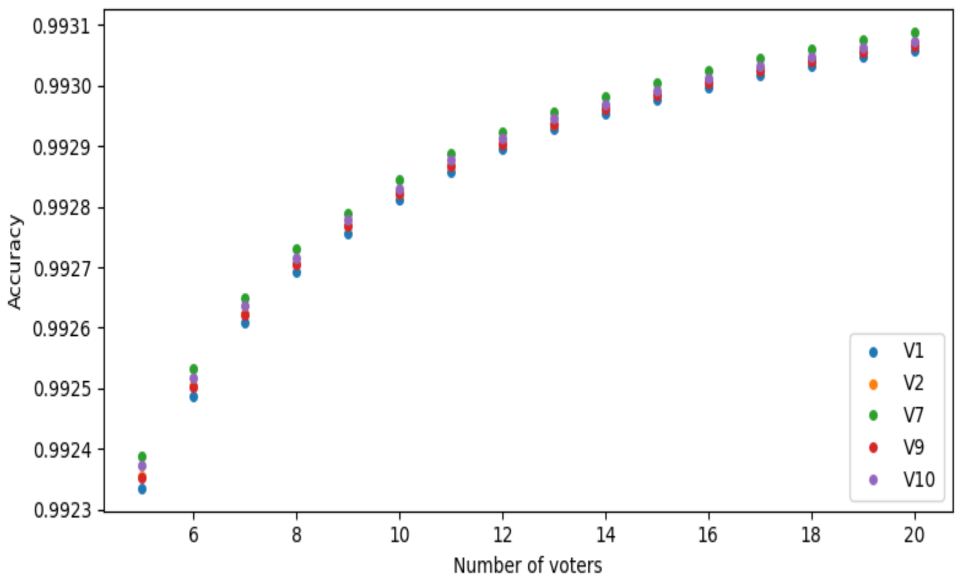


Figure 36: The performance results of our algorithm with V1 fuzzy average voting function by $5-20$ voters on average on test data using different parameters for the fuzzification of the training data class membership values.

As we can see in figure 36 the accuracy of each of the depicted voting functions grows similarly with the increasing number of voters. With a lower size of the committee, there were somewhat larger differences between their performance, with a higher number of voters the difference was smaller.

8.3.2 Voting experiment 2 with our algorithm based on a convolutional neural network [65]

Our second committee machine experiment on voting schemes has been run with 1000 sessions. In each turn, $5-20$ learners voted with the voting functions (V1-V10) described above.

voting function	**MIN**	**AVG**	**MAX**
min(accuracy)	0.992700	0.995422	0.996900
avg(accuracy)	0.995188	0.996306	0.997260
max(accuracy)	0.995600	0.997043	0.998000
V1–fuzzy average	0.996000	0.997351	0.998400
V2–weighted by confidence	0.995900	0.997360	0.998300
V3–weighted by diff from V1	0.996000	0.997346	0.998300
V4–weighted by 1/failures	0.995500	0.997321	0.998300
V5–plurality voting	0.995500	0.997280	0.998400
V6–Borda voting	0.995600	0.997296	0.998400
V7–geometric mean voting	0.996000	**0.997367**	0.998300
V8–meta–plurality (V1-V7)	0.995900	0.997353	0.998400
V9–meta–plurality (V1-V4, V7)	0.996000	0.997356	0.998400
V10–meta–fuzzy avg (V1-V4, V7)	0.996100	0.997350	0.998300

Table 3: Minimum, maximum, and average accuracy on the test samples for ensemble learning voting experiment 2

Table 3 shows the accumulated results of the tested voting functions with the minimum, average, and maximum number of the failed samples of the individual learners included. The best individual result was 20 fails on 10000 test samples, the worst was 73 failed samples and on average they performed as low as 36.94 fails from 10000 samples as individual learners. The well-known fuzzy voting performed 26.49 fails on average. There were no big differences among the voting functions, the best result was given by the product voting (i.e. geometric mean–V7) from committee results on training failures.

For some of the best performing voting functions, we also show the accuracy achieved by them with different number of voters.

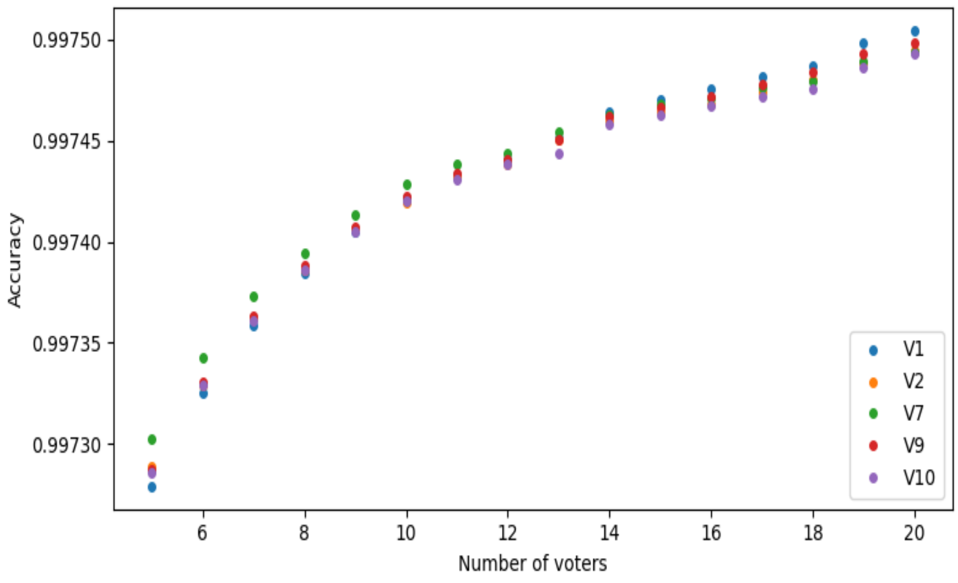


Figure 37: The performance results of our algorithm with V1 fuzzy average voting function by $5-20$ voters on average on test data using different parameters for the fuzzification of the training data class membership values.

As we can see in Figure 37 the five voting functions show similar behavior. All of them resulted in better accuracy with more voters. There were voting functions that performed better with the lower number of voters, while others gave higher accuracy with increasing the number of voters. This behavior can be further examined in the future since that can mean, that for specific problems different voting functions might be chosen for better performance in case of the different committee sizes.

8.3.3 Voting experiment 3 with an algorithm based on the squeeze-and-excite algorithm

The third experiment ran also 1000 times on a modified version of [58]. It was executed with 20 epochs for each learner.

voting function	**MIN**	**AVG**	**MAX**
min(accuracy)	0.996200	0.996878	0.997700
avg(accuracy)	0.996840	0.997418	0.997886
max(accuracy)	0.997100	0.997914	0.998500
V1–fuzzy average	0.997300	0.998126	0.998700
V2–weighted by confidence	0.997300	0.998122	0.998700
V3–weighted by diff from V1	0.997200	0.998128	0.998700
V4–weighted by 1/failures	0.997300	0.998126	0.998700
V5–plurality voting	0.997000	0.998044	0.998700
V6–Borda voting	0.997000	0.998044	0.998700
V7–geometric mean voting	0.997100	0.998126	0.998700
V8–meta–plurality (V1-V7)	0.997200	0.998125	0.998700
V9–meta–plurality (V1-V4, V7)	0.997200	0.998128	0.998700
V10–meta–fuzzy avg (V1-V4, V7)	0.997200	**0.998129**	0.998700

Table 4: Minimum, maximum, and average accuracy for ensemble learning voting experiment 3

Table 4 shows the results where $5 - 20$ voters voted in each turn using the above-defined voter functions. The best individual result was 15 fails from 10000 test samples, the worst individual result was 38 failed samples and on average they performed only 25.82 fails as individual learners. The well-known fuzzy voting performed 18.74 on average. There were no big differences among the voting functions, the best result (18.71 fails) came from our meta fuzzy voter function (V10).

For chosen voting functions (V1, V2, V7, V9, V10) we also show the accuracy achieved by them with different number of voters.

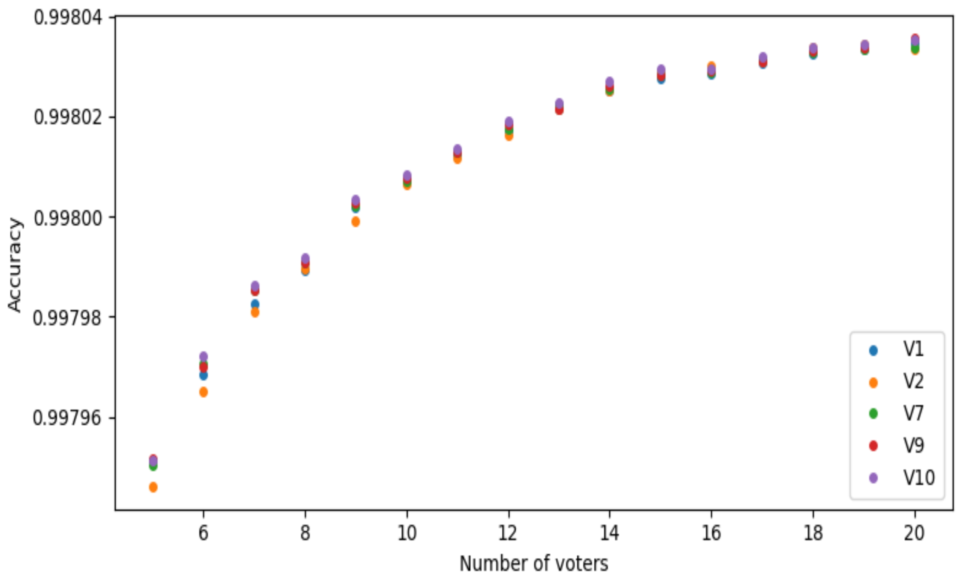


Figure 38: The performance results of our algorithm with V1 fuzzy average voting function by $5-20$ voters on average on test data using different parameters for the fuzzification of the training data class membership values.

As we can see in Figure 38 the behavior of the chosen voting functions was similar. The accuracy of them grows similarly with the increasing number of voters. With lower committee size there were more significant differences between their accuracy, this difference is lower when the number of voters increases.

Although the difference with larger committees is not significant, this behavior can be interesting for machine learning environments on lower performance hardware, where only smaller ensembles can be executed.

8.3.4 Correlation of variables

Since we have performed multiple experiments that gave a large number of results, these data can also serve for subsequent researches.

For that purpose, we also provide a correlation matrix that shows the correlation between the measured variables, i.e. the performance of the individual learners and the voting functions V1-V10.

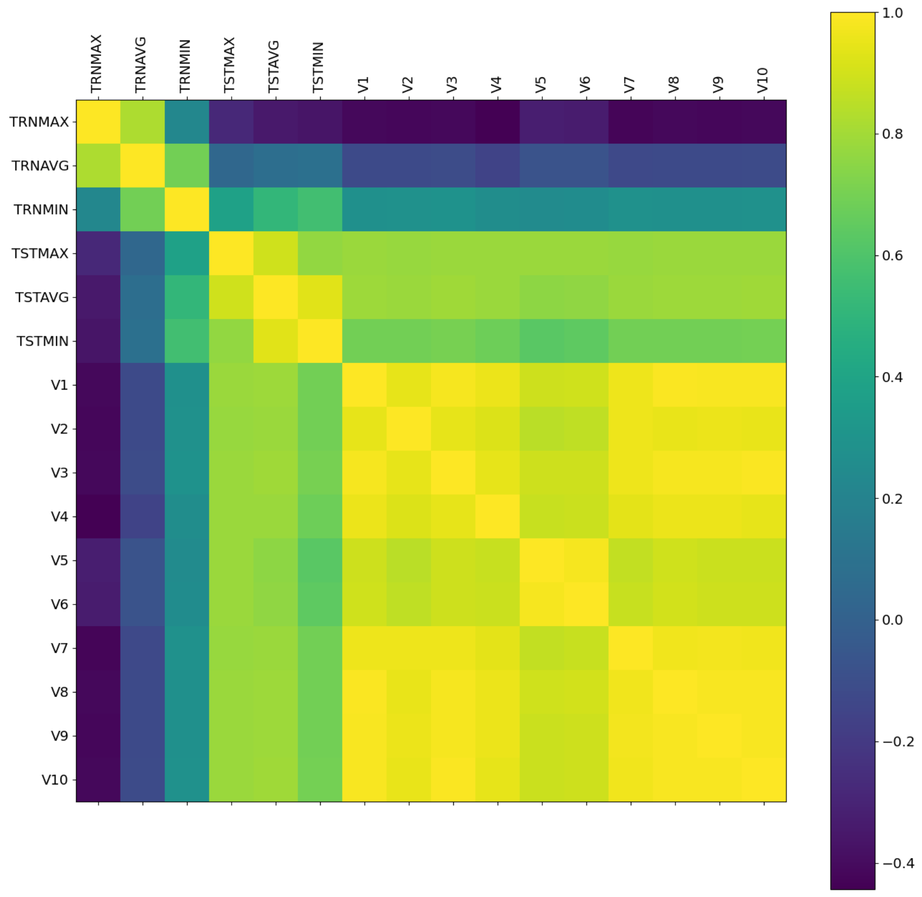


Figure 39: The correlation (corrcoef) between the variables MAX, AVG, MIN accuracy of individual learners on training and test data, and V1-V10 voting functions

As can be seen in Figure 39, the correlation can be different between the measured variables. As we can expect, it is stronger between the fuzzy voting function variants; and stronger between the plurality and the Borda voting too, and it is weaker between fuzzy and plurality voters. This correlation between voting functions can be important for the selection of voting functions for meta voting functions.

When we examine the correlation, we can find that the TSTMAX has a stronger correlation with the accuracy of the voting functions, compared to TSTMIN. One obvious reason for that can be that with increasing the size of the ensemble, the probability to have learners achieving lower performance is higher, but it also leads to better accuracy of the committee machine, as we described previously.

It is also interesting to check the correlation of the training accuracy with the voting functions. As we can see, the TRNMAX has a negative correlation with them. It probably means that it is the effect of overfitting since TRNMAX has a negative correlation also with TSTAVG, so the learners having top accuracy on training data negatively affect the average test accuracy. This justifies the importance of introducing a stopping condition, or generalization methods.

9 Conclusions and future work

In our research we successfully developed fast (short-cut) evaluation algorithms for expression trees of three main fuzzy logic types, the Gödel fuzzy logic, the Product fuzzy logic, and the Łukasiewicz fuzzy logic. The presented results show that using these algorithms the evaluation of large expression trees can be much faster, the majority of the nodes can be cut from the evaluation saving time and other resources.

Further research and development can be done to improve the fast evaluation algorithm to include the weak and strong conjunction and disjunction operators in the same algorithm, and to develop it for other fuzzy logic types, as well.

From the results of our fuzzification experiments, we can conclude that the fuzzification of the training data binary class membership values can improve the accuracy of the predictions for both individual learners and multiple learners as an ensemble method.

The performance of this method can be different using it with the various machine learning algorithms. For algorithms, which don't have peak performance, our method gave better improvement. With top performance algorithm variants, that can probably already better handle inaccuracies in training data, the advantage was smaller.

The research and development on this can be continued possibly to make decisions during run-time, e.g. when to start fuzzification and with what factors.

As a conclusion regarding the voting functions, we discovered that there is no voting function which is always the winner. The availability of multiple voting functions can however lead to better performance if the best performer function will be chosen for a specific problem set. Larger differences were encountered with smaller committees, this means, that our research might be more useful for lower performance environments.

Some of the voting functions, including the ones proposed in our research, resulted in better accuracy in our experiments, compared to the most frequently used well-known fuzzy average and plurality voting functions (V2, V7, V9, V10).

Future work should be performed in this research to further analyze the behavior of the voting functions, with the goal to make automatic decisions on choosing a good voting function for the specific problem.

10 Summary

In recent decades, the increase in computing capacity, and the availability of large amounts of data, the Internet, and sensors (including image capture devices), the development of machine learning algorithms have brought the need and the opportunity to apply ever-increasing levels of artificial intelligence into our everyday lives. Fuzzy logic and neural networks also play a significant role in decision support systems, in the dissertation, we discuss our efficient, new solutions developed for these areas.

After the introduction and the description of the motivations, chapter 2 briefly describes the objectives and results achieved in the field of rapid evaluation of Gödel, Product, and Łukasiewicz fuzzy logic expression trees, as well as further developments, to improve the performance of neural network algorithms.

Chapter 3 describes the basics required for the presented results, from Boolean logic expressions to fuzzy logic expression trees, and then describes the three main fuzzy logics for which fast evaluation algorithms were developed.

The introduction to neural networks was followed by an introduction to the committee machines, and finally a summary of some well-known ensemble methods.

Chapter 4 describes the generation of formula trees used for measurements, which was necessary for the analyses related to the algorithms presented in the next chapter. The formula tree generator can be flexibly parameterized to be suitable for the production of formulas of various shapes with adjustable values for all three fuzzy logic involved.

Chapter 5 shows the new algorithms developed for evaluating fuzzy logic formula trees in three subchapters, according to the rules of Gödel fuzzy logic, Product fuzzy logic, and finally, Łukasiewicz fuzzy logic.

In chapter 6, we present the results of measurements from a large number of simulations run with algorithms and their analysis. The simulations run with fast evaluation algorithms on Gödel, Product, and Łukasiewicz fuzzy logic expression trees, generated with different parameters for various expression tree shapes and value distributions. The results show high efficiency.

Chapter 7 shows our new algorithm for increasing the efficiency of neural networks, first the algorithm for fuzzification of binary class membership values of the training data, followed by our new voting functions and meta voting functions for committee machines.

In chapter 8, we present the results of the algorithms, developed in our research on neural network algorithms, based on a large number of learning measurements.

From measuring the performance of our algorithms developed for fast evaluation of fuzzy logic expression trees, we can conclude that the improvement is very significant compared to simple evaluation. Furthermore, we have also shown that the cutting possibilities of binary logic have been extended, with a significant increase in performance compared to them.

From our research, we can conclude that the fuzzification of the binary class membership values of the training data can improve the accuracy of the predictions. For algorithms that do not have peak performance, our method had stronger improvement. With the high-performance algorithm variants, which are probably already better able to handle training data inaccuracies, the advantage was smaller.

As a conclusion about voting functions, we found that there is no voting function that is always that winner. However, the availability of multiple voting features can lead to better performance if the function with the best performance is selected for a given set of problems. Some of the proposed voting features had better accuracy in our experiments compared to the most commonly used fuzzy mean and plurality voting functions.

11. Összefoglalás

Az elmúlt évtizedekben a számítási kapacitás növekedése, az adatok nagy tömegének rendelkezésre állása, az internet és a szenzorok (ideértve a képrögzítő eszközöket is), a gépi tanuló algoritmusok fejlődése a mindennapjainkba is elhozták az egyre magasabb szintű mesterséges intelligencia alkalmazásának igényét és lehetőségét. A döntéstámogató rendszerekben a fuzzy logika és a neurális hálózatok is jelentős szerepet kapnak, a disszertációban ezen területekre fejlesztett hatékony, új megoldásainkról értekezünk.

A dolgozatban, a bevezetés és a motivációk leírása után, a 2. fejezet röviden ismertette a célkitűzéseket és az elért eredményeket mind a Gödel, a Product, valamint a Łukasiewicz fuzzy logika kifejezésfák gyors kiértékelése, mind pedig a neurális hálózat algoritmusok teljesítményének javítását célzó továbbfejlesztések terén.

A 3. fejezet ismerteti a bemutatott eredményekhez szükséges alapokat, a Boolean logika kifejezésektől a fuzzy logika kifejezésfákig, majd ismertet három fő fuzzy logikát, amelyekre a gyors kiértékelő algoritmusok készültek. A neurális hálózatokba történő bevezetés után a bizottság-gépek ismertetése, végül néhány ismertebb együttes (ensemble) módszer ismertetése következett.

A 4. fejezetben a fuzzy logika gyors kiértékelő algoritmusokkal kapcsolatos mérésekhez, vizsgálatokhoz használt formula fák generálásának leírása következett, amely szükségeltetett a következő fejezetben bemutatott algoritmusokkal kapcsolatos analízisekhez.

A formula fa generátor rugalmasan paraméterezhető, hogy mindhárom érintett fuzzy logikára szabályozható értékekkel rendelkező, szabályozható alakú formula fák előállítására alkalmas legyen.

Az 5. fejezetben mutatjuk be a fuzzy logika formula fák kiértékelésére kifejlesztett új algoritmusokat három alfejezetben, sorrendben a Gödel fuzzy logika, a Product fuzzy logika, majd végül a Łukasiewicz fuzzy logika szabályainak megfelelően.

A 6. fejezet prezentálja az algoritmusokkal futtatott nagy számú szimulációról készült mérések eredményeit, és azok analízisét. A Gödel, a Product, valamint a Łukasiewicz fuzzy logika kifejezésfák gyors kiértékelésére futtatott szimulációkkal, különböző paraméterekkel generált, változatos alakú kifejezésfák esetében mutatjuk be az eredményeket, melyek magas hatékonyságot mutatnak.

A 7. fejezetben bemutatjuk a neurális hálózatok hatékonyságát növelő új fuzzifikáló algoritmusunkat a tanulási adatokhoz megadott bináris osztálytagsági értékek korrekciójára, valamint a bizottság-gépekhez használható új szavazó függvényeinket és meta szavazó függvényeinket.

A 8. fejezet a neurális hálózat algoritmusokkal kapcsolatos kutatásunk során alkotott új algoritmus, valamint új szavazó függvények alkalmazásával kapott futtatási eredményeket mutatja be nagyszámú tanulás során folytatott mérések alapján.

Kutatásunk alapján a fuzzy logika kifejezésfák gyors kiértékelésére fejlesztett algoritmusaink teljesítményének méréséből megállapíthatjuk, hogy az egyszerű kiértékeléshez viszonyítva nagyon jelentős a javulás. Továbbá megmutattuk azt is, hogy a bináris logika vágási lehetőségeit sikeresen kiterjesztettük, azokhoz képest is számottevő a teljesítmény növekedése.

A neurális hálózatokkal kapcsolatos kutatásunkból arra a következtetésre juthatunk, hogy a tanulási adatok bináris osztálytagsági értékeinek fuzzifikálása javíthatja az előrejelzések pontosságát. Azon algoritmusok esetében, amelyek nem rendelkeznek csúcsteljesítménnyel, a módszerünk erőteljesebben javította az eredményeket. A nagy teljesítményű algoritmus variánsokkal, amelyek valószínűleg már jobban képesek kezelni a tanulási pontatlanságait, a javulás kisebb mértékű volt.

A bizottság-gép szavazó függvényekkel kapcsolatos következtetésként azt tapasztaltuk, hogy nincs olyan szavazási funkció, amely mindig jobb a többinél. A több szavazási funkció elérhetősége azonban jobb teljesítményhez vezethet, ha a legjobb funkciót választja egy adott problémakörre. Biztató eredmény, hogy a javasolt szavazási funkciók közül többnek a pontossága jobb volt a kísérleteinkben, összehasonlítva a leggyakrabban használt fuzzy mean és plurality szavazási funkciókkal.

12 Publications

Journal publications related to the dissertation

[1] B. Nagy, R. Basbous, and T. Tajti. “Lazy evaluations in Łukasiewicz type fuzzy logic”. In: *Fuzzy Sets and Systems* 376 (2019), pp. 127–151.

[2] T. Tajti. “Fuzzification of training data class membership binary values for neural network algorithms”. In: *Annales Mathematicae et Informaticae* 52 (2020), pages to be approved.

[3] T. Tajti. “New voting functions for neural network algorithms”. In: *Annales Mathematicae et Informaticae* 52 (2020), pages to be approved.

Foreign language conference proceedings related to the dissertation

[4] R. Basbous, B. Nagy, and T. Tajti. “Short Circuit Evaluations in Gödel Type Logic”. In: *Proc. of FANCCO 2015: 5th International Conference on Fuzzy and Neuro Computing, Advances in Intelligent Systems and Computing.* Vol. 415. 2015, pp. 119–138.

[5] R. Basbous, T. Tajti, and B. Nagy. “Fast Evaluations in Product Logic: Various Pruning Techniques”. In: *FUZZ-IEEE 2016 - the 2016 IEEE International Conference on Fuzzy Systems.* Vancouver, Canada: IEEE, 2016, pp. 140–147.

[6] C. Biró, G. Kusper, and T. Tajti. “How to generate weakly nondecisive SAT instances”. en. In: *Intelligent Systems and Informatics (SISY), 2013 IEEE 11th International Symposium on : IEEE 11th International Symposium on Intelligent Systems and Informatics : proceedings.* Budapest, Magyarország: IEEE Hungary, 2013, pp. 265–269.

Other publications

[7] T. Tajti. “Fuzzification of neural network pattern outputs for classification problems [abstract]”. ICAI 2020 : 11th International Conference on Applied Informatics, Eger, Hungary. 2020.

[8] Z. Gál, T. Tajti, and G. Terdik. “Surprise event detection of the supercomputer execution queues”. en. In: *Annales Mathematicae et Informaticae* 44 (2015), pp. 87–97.

[9] Z. Gál et al. “Performance evaluation of massively parallel communication sessions”. en. In: *The Sixth International Conference on Parallel, Distributed, GPU and Cloud Computing for Engineering.* Civil-Comp Press, 2019, pp. 1–19.

[10] T. Tajti and B. Nagy. “Motion sensor data correction using multiple sensors and multiple measurements”. en. In: *SAMI 2016 : IEEE 14th International Symposium on Applied Machine Intelligence and Informatics, New York (NY), Amerikai Egyesült Államok.* IEEE, 2016, pp. 287–291.

[11] T. Tajti et al. “Indoor localization using NFC and mobile sensor data corrected using neural net”. en. In: *ICAI 2014: Proceedings of the 9th International Conference on Applied Informatics.* Vol. 1-2. Eszterházy Károly Tanárképzõ Fõiskola, 2014, pp. 163–169.

[12] T. Gregus ; G. Geda ; P. Magyar ; T. Tajti ; A. Perjési. “Business Oriented Creature Identification”. en. In: *ICAI 2014 The 9th International Conference on Applied Informatics to be held in Eger.* Eger, Magyarország: Eszterházy Károly College, 2014.

[13] C. Biró et al. *Look-ahead alapú SAT solver-ek párhuzamosíthatóságának vizsgálata.* hu. Budapest, Magyarország, 2013.

[14] G. Kusper, C. Biró, and T. Tajti. *WnDGen: Weakly Nondecisive SAT Problem Generator.* en. Tudományos célú szoftver, Version: 1.0, 29.05.2012, Current version: 2.0, 07.01.2013, Megjelenés: Magyarország. 2013.

[15] G. Kusper, C. Biró, and T. Tajti. *SATCounter: Count Clear Clauses SAT Solver.* en. Vol. 1. Magyarország, 2013.

[16] Z. Gál and T. Tajti. “Complex event processing in supercomputer environment: sensor and neural network based analysis”. en. In: *IEEE 4th International Conference on Cognitive Infocommunications: CogInfoCom 2013, New York (NY), Amerikai Egyesült Államok.* IEEE, 2013, pp. 735–740.

[17] J. Kajdi ; K. Nagy ; Ő. Legeza ; J. Kövesi ; T. Tajti ; A. Vigvári. *Az önkormányzati adósságregiszter kialakításának megalapozása.* hu. Magyar Közigazgatási Intézet, 2001, pp. 197–283.

13 References

[1] C. Biró, G. Kusper, and T. Tajti. “How to generate weakly nondecisive SAT instances”. en. In: *Intelligent Systems and Informatics (SISY), 2013 IEEE 11th International Symposium on : IEEE 11th International Symposium on Intelligent Systems and Informatics : proceedings*. Budapest, Magyarország: IEEE Hungary, 2013, pp. 265–269.

[2] R. Basbous, B. Nagy, and T. Tajti. “Short Circuit Evaluations in Gödel Type Logic”. In: *Proc. of FANCCO 2015: 5th International Conference on Fuzzy and Neuro Computing, Advances in Intelligent Systems and Computing*. Vol. 415. 2015, pp. 119–138.

[3] R. Basbous, T. Tajti, and B. Nagy. “Fast Evaluations in Product Logic: Various Pruning Techniques”. In: *FUZZ-IEEE 2016 - the 2016 IEEE International Conference on Fuzzy Systems*. Vancouver, Canada: IEEE, 2016, pp. 140–147.

[4] B. Nagy, R. Basbous, and T. Tajti. “Lazy evaluations in Łukasiewicz type fuzzy logic”. In: *Fuzzy Sets and Systems* 376 (2019), pp. 127–151.

[5] T. Tajti. “Fuzzification of neural network pattern outputs for classification problems [abstract]”. ICAI 2020 : 11th International Conference on Applied Informatics, Eger, Hungary. 2020.

[6] T. Tajti. “Fuzzification of training data class membership binary values for neural network algorithms”. In: *Annales Mathematicae et Informaticae* 52 (2020), pages to be approved.

[7] T. Tajti. “New voting functions for neural network algorithms”. In: *Annales Mathematicae et Informaticae* 52 (2020), pages to be approved.

[8] S. Russell and P. Norvig. *Artificial Intelligence, a Modern Approach*. en. New Jersey: Prentice-Hall, 2003.

[9] E. Rich and K. Knight. *Artificial Intelligence*. en. New York: McGraw-Hill Inc, 1991.

[10] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. 2nd. USA: Prentice Hall PTR, 1998.

[11] Shaohua Wan and Hua Yang. “Comparison among Methods of Ensemble Learning”. In: *2013 International Symposium on Biometrics and Security Technologies* (2013), pp. 286–290.

[12] D. Opitz and R. Maclin. “Popular ensemble methods”. en. In: *An empirical study Journal of Artificial Intelligence Research* 11 (1999), pp. 169–198.

[13] R. Basbous and B. Nagy. “Strategies to Fast Evaluation of Tree Networks, Acta Polytechnica Hungarica”. en. In: *Acta Polytechnica Hungarica* 12.6 (2015), pp. 127–148.

[14] J. Bell and M. Machover. *A Course In Mathematical Logic, North-Holland.* en. New York and Oxford, 1977.

[15] R. Basbous and B. Nagy. “Generalized Game Trees and their Evaluation”. en. In: *Proc. of CogInfoCom 2014: 5th IEEE International Conference on Cognitive Infocommunications.* Italy: Vietri sul Mare, 2014, pp. 55–60.

[16] L. Zadeh. “Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A”. en. In: *Advances in Fuzzy Systems-Applications and Theory.* Ed. by Zadeh. Vol. 6. River Edge, NJ, USA: World Scientific, 1996.

[17] P. Hájek. “Metamathematics of Fuzzy Logic”. nl. In: *Trends in Logic* 4 (1998).

[18] L.A. Zadeh. “Fuzzy sets”. In: *Information and Control* 8.3 (1965), pp. 338–353.

[19] L. Zadeh. *Fuzzy logic-a Personal Perspective, Fuzzy Sets and Systems.* en. 2015.

[20] R. Hähnle. “Many-valued logic and mixed integer programming”. en. In: *Annals of Mathematics and Artificial Intelligence* 12 (1994), pp. 231–264.

[21] K. Gödel. “Zum intuitionistische Aussagenkalkül”. In: *Mathematish-Naturwissenschaftliche Klasse* 69 (1932). reprinted in Kurt Gödel, Collected Works, Oxford University Press, 1985, pp. 65–66.

[22] S. Gottwald. *Many-Valued Logic.* en. Edward N. Zalta (ed.) The Stanford Encyclopedia of Philosophy, URL=http://plato.stanford.edu/entries/logic-manyvalued. 2015.

[23] Matthias Baaz, Norbert Preining, and Richard Zach. “First-order Gödel logics”. In: *Annals of Pure and Applied Logic* 147.1-2 (2007), pp. 23–47.

[24] B. Nagy. “A General Fuzzy Logic Using Intervals”. en. In: *Proc of. 6th International Symposium of Hungarian Researchers on Computational Intelligence.* Budapest, Hungary, 2005, pp. 613–624.

[25] P. Hájek, L. Godo, and F. Esteva. “A complete many-valued logic With product-conjunction”. en. In: *Archive for Mathematical Logic* 35 (1996), pp. 191–208.

[26] R.J. Adillon and V. Verdu. “On product logic”. en. In: *Soft Computing* 2.ue 3 (1998), pp. 141–146.

[27] M. Baaz et al. “Embedding logics into product logic”. pt. In: *Studia Logica* 61 (1998), pp. 35–47.

[28] G. Metcalfe, N. Olivetti, and D. Gabbay. “Analytic Calculi for Product Logics”. en. In: *Archive for Mathematical Logic* 43 (2004), pp. 859–889.

[29] G. Metcalfe, N. Olivetti, and D. Gabbay. *Proof Theory for Fuzzy Logics.* en. Vol. 36. Springer-Verlag, 2008.

[30] B. Nagy. “Many-valued logics and the logic of the C programming language”. en. In: *Proc. of ITI 2005: 27th International Conference on Information Technology Interfaces (IEEE), Cavtat, Croatia.* 2005, pp. 657–662.

[31] J. Lukasiewicz and A. Tarski. “Investigations in the Sentential Calculus”. en. In: (1930). translation in J. Lukasiewicz, Selected Works, Reidel, Dordrecht (1970).

[32] S. Kundu and J. Chen. *Fuzzy Logic or Łukasiewicz Logic: A Clarification, Fuzzy Sets and Systems.* en. 1998.

[33] S. Aguzzoli and A. Ciabattoni. “Finiteness in infinite-valued Łukasiewicz logic”. et. In: *Journal of Logic, Language, and Information* 9 (2000), pp. 5–29.

[34] Y. Lecun et al. “Gradient-based learning applied to document recognition”. en. In: *Proc. IEEE* 86 (1998), pp. 2278–2324.

[35] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. “Multi-column deep neural networks for image classification”. In: *2012 IEEE conference on computer vision and pattern recognition.* IEEE. 2012, pp. 3642–3649.

[36] Alexis Conneau et al. “Very deep convolutional networks for text classification”. In: *arXiv preprint arXiv:1606.01781* (2016).

[37] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. “Conditional time series forecasting with convolutional neural networks”. In: *arXiv preprint arXiv:1703.04691* (2017).

[38] Volker Tresp. “The generalized Bayesian committee machine”. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’00.* ACM Press, 2000.

[39] Yoav Freund. “Boosting a weak learning algorithm by majority”. In: *Information and computation* 121.2 (1995), pp. 256–285.

[40] Volker Tresp. “Committee machines”. In: *Handbook for neural network signal processing* (2001), pp. 1–18.

[41] G. Fumera and F. Roli. "A theoretical and experimental analysis of linear combiners for multiple classifier systems". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.6 (2005), pp. 942–956.

[42] Kamran Kowsari et al. "Rmdl: Random multimodel deep learning for classification". In: *Proceedings of the 2nd International Conference on Information System and Data Mining.* 2018, pp. 19–28.

[43] Galina Rogova. "Combining the results of several neural network classifiers". In: *Neural Networks* 7.5 (1994), pp. 777–781.

[44] G. Brawn. "Ensemble Learning". In: *Encyclopedia of Machine Learning.* Springer US, 2011, pp. 312–320.

[45] Whitman Richards, H Sebastian Seung, and Galen Pickard. "Neural voting machines". In: *Neural Networks* 19.8 (2006), pp. 1161–1167.

[46] Gasser Auda, Mohamed Kamel, and Hazem Raafat. "Voting schemes for cooperative neural network classifiers". In: *Proceedings of ICNN'95-International Conference on Neural Networks.* Vol. 3. IEEE. 1995, pp. 1240–1243.

[47] Ludmila I Kuncheva. "A theoretical study on six classifier fusion strategies". In: *IEEE Transactions on pattern analysis and machine intelligence* 24.2 (2002), pp. 281–286.

[48] Ury Naftaly, Nathan Intrator, and David Horn. "Optimal ensemble averaging of neural networks". In: *Network: Computation in Neural Systems* 8.3 (1997), pp. 283–296.

[49] B. Efron. "Bootstrap methods: another look at the jackknife". en. In: *The Annals of Statistics* 7.1 (1979), pp. 1–26.

[50] C. Shen and H. Li. "On the dual formulation of boosting algorithms". en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.12 (2010), pp. 2216–2231.

[51] D.H. Wolpert. "Stacked Generalization". de. In: *Neural Networks* 5.2 (1992), pp. 241–259.

[52] L. Breiman. "Random forests". en. In: *Machine Learning* 45.1 (2001), pp. 5–32.

[53] David F. Nettleton, Albert Orriols-Puig, and Albert Fornells. "A study of the effect of different types of noise on the precision of supervised learning techniques". In: *Artificial Intelligence Review* 33.4 (2010), pp. 275–306.

[54] M. Pechenizkiy et al. “Class Noise and Supervised Learning in Medical Domains: The Effect of Feature Extraction”. In: *19th IEEE Symposium on Computer-Based Medical Systems (CBMS’06)*. 2006, pp. 708–713.

[55] Xingquan Zhu and Xindong Wu. “Class Noise vs. Attribute Noise: A Quantitative Study”. In: *Artificial Intelligence Review* 22.3 (2004), pp. 177–210.

[56] Yann LeCun, Corinna Cortes, and Christopher JC Burges. “The MNIST database of handwritten digits, 1998”. In: *URL http://yann.lecun.com/exdb/mnist* 10.34 (1998), p. 14.

[57] Jie Hu, Li Shen, and Gang Sun. “Squeeze-and-excitation networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7132–7141.

[58] Matuzas77. *MNIST classifier with average 0.17% error*. https://github.com/Matuzas77/MNIST-0.17/blob/master/MNIST_final_solution.ipynb. 2020 (last accessed October 30, 2020).

[59] J. Siltaneva. “A Comparison of Random Binary Tree Generators”. In: *The Computer Journal* 45.6 (2002), pp. 653–660.

[60] D. E. Knuth and R. W. Moore. “An analysis of alpha-beta pruning”. es. In: *Artificial Intelligence* (1975), pp. 6, 293–326.

[61] Robert Fullér. “Fuzzy systems”. In: *Introduction to Neuro-Fuzzy Systems*. Springer, 2000, pp. 1–131.

[62] Sebastien C Wong et al. “Understanding data augmentation for classification: when to warp?” In: *2016 international conference on digital image computing: techniques and applications (DICTA)*. IEEE. 2016, pp. 1–6.

[63] Florian Tramèr et al. “Ensemble adversarial training: Attacks and defenses”. In: *arXiv preprint arXiv:1705.07204* (2017).

[64] Leijun Li et al. “Exploration of classification confidence in ensemble learning”. In: *Pattern recognition* 47.9 (2014), pp. 3120–3131.

[65] Chris Deotte. *25 Million Images! [0.99757] MNIST*. https://www.kaggle.com/cdeotte/25-million-images-0-99757-mnist. 2020 (last accessed October 30, 2020).

[66] Louisa Lam and Ching Y Suen. “Optimal combinations of pattern classifiers”. In: *Pattern Recognition Letters* 16.9 (1995), pp. 945–954.

14 Appendix

For better readability some of the proofs about the algorithms are moved to the appendix. We also recall the lemmas themselves.

Lemma 1. Both PruneEval1 and PruneEval2 for Łukasiewicz fuzzy logic formula tree evaluation (Algorithm 5 and Algorithm 6 respectively) use variables *lower*, *upper* and v such that they always have values with condition $0 \leq lower \leq 1, 0 \leq upper \leq 1, 0 \leq v \leq 1$ during the recursive calls.

Proof. First, we note that *lower* and *upper* are having the values $lower = 0$ and $upper = 1$ at the initial call in both of the algorithms. The proof goes by induction on the height of the formula tree.

- The base case is with height 0, i.e., for formulae without connectives. If such a formula is evaluated with parameters $0 \leq lower \leq 1, 0 \leq upper \leq 1$, then it gets value without any recursive calls, i.e., it either gets the value of *lower* (in case of cut) or its real value. However, in both cases, the condition $0 \leq v \leq 1$ definitely holds.

- Let the induction hypothesis be formulated as follows. Assume that both algorithms use $0 \leq lower, upper, v \leq 1$ at each recursive call for all formulae having height at most for any fixed non-negative integer k if at the initial call the condition $0 \leq lower, upper \leq 1$ meets.

- Let us prove the inheritance, i.e., the statement that the conditions $0 \leq lower \leq 1, 0 \leq upper \leq 1, 0 \leq v \leq 1$ hold also for formulae with height $k + 1$

  ◦ When the formula is evaluated by any of the algorithms with parameters $0 \leq lower = upper \leq 1$, then $v = lower$ is assigned and, thus, also $0 \leq v \leq 1$ is satisfied. Further, in the proof, we assume that $lower \neq upper$ (i.e., we do not mention again the case of equality).

  There are various cases by the connective given in the root of the expression.

  o If the root is a negation node, then in both algorithms, $lower_1 = 1 - upper$ and $upper_1 = 1 - lower$ are used to evaluate its child. If any of the algorithms are called with the condition $0 \leq lower \leq 1, 0 \leq upper \leq 1$, then $0 \leq lower_1 \leq 1, 0 \leq upper_1 \leq 1$ also hold, moreover the height of the formula-tree rooted at the child is exactly k. Thus, applying the hypothesis, at each recursive call during the evaluation the variables $lower, upper$ have values that are always between 0 and 1 (inclusively, i.e., allowing to have value 0 or 1 also, moreover each evaluated node x gets also value v_x such that $0 \leq v_x \leq 1$. Consequently, the root gets the value $v = 1 - v_1$ in both algorithms, where v_1 is the value of the child. Since $0 \leq v_1 \leq 1$ implies $0 \leq v \leq 1$, the inheritance in this case is proven.

o If the root is a conjunction node, then both children are roots of sub-trees at most height k. One of the children is evaluated by $lower_1 = lower$, and $upper_1 = 1$. Subsequently, if $0 \leq lower \leq 1$, then $0 \leq lower_1 \leq 1, 0 \leq upper_1 \leq 1$ also hold, thus, by the induction hypothesis, all subsequent calls for the formula rooted in this child uses parameters *lower* and *upper* with the condition of the lemma, and also value v_x always fulfills $0 \leq v_x \leq 1$, for every evaluated node of this sub-tree. Thus, also $0 \leq v_1 \leq 1$ is satisfied. Then, the other child is evaluated by parameters $lower_2 = min\{1+lowerv_1, 1\}$ and $upper_2 = min\{1 + upper - v_1, 1\}$ in both algorithms. Knowing that $0 \leq lower \leq 1, 0 \leq upper \leq 1and0 \leq v_1 \leq 1$, it is clear that $0 \leq lower_2 \leq 1, 0 \leq upper_2 \leq 1$. By applying the hypothesis the sub-tree of this child is evaluated in such a way that
$0 \leq lower \leq 1, 0 \leq upper \leq 1$ is always satisfied, and each evaluated node x gets a value v_x such that $0 \leq v_x \leq 1$. Consequently, also this child gets a value v_2 according to this condition. Then, the value v is computed as $v = max\{v_1 + v_2 - 1, 0\}$, which inherits the property $0 \leq v \leq 1$ from v_1 and v_2.

o In case of disjunction at the root, again both children have sub-formulae with height at most k. In both algorithms, one of the children is a called with parameters $lower_1 = 0andupper_1 = upper$. If $0 \leq upper \leq 1, then 0 \leq lower_1 \leq 1, 0 \leq upper_1 \leq 1$ also hold. By the hypothesis during the evaluation of this subformula the conditions $0 \leq lower \leq 1, 0 \leq upper \leq 1, 0 \leq v \leq 1$ hold. Especially, $0 \leq v_1 \leq 1$ also holds. The evaluation of the other child is called with $lower_2 = max\{lower - v_1, 0\}$ and $upper_2 = max\{upper - v_1, 0\}$ in both algorithms. By
$0 \leq lower \leq 1, 0 \leq upper \leq 1, 0 \leq v_1 \leq 1$, the conditions $0 \leq lower_2 \leq 1, 0 \leq upper_2 \leq 1$ also hold. Thus, the hypothesis can be applied for the sub-tree rooted in this child. Consequently, the value of this child v_2 also satisfies $0 \leq v_2 \leq 1$. Then, value $v = min\{v_1 + v_2, 1\}$ is computed, which satisfies $0 \leq v \leq 1$.

o If the root is an implication node, let us analyze first Algorithm 5.

The first child is called with parameters $lower_1 = 1 - upper$, and $upper_1 = 1$. The condition $0 \leq upper \leq 1$ implies that both $0 \leq lower_1 \leq 1$ and $0 \leq upper_1 \leq 1$, thus the hypothesis is applicable to the subformula rooted in the first child. Therefore, all the values $lower$, $upper$, and v during the evaluation have values between 0 and 1 (inclusively), which also holds for v_1.
Then, the second child is evaluated with parameters $lower_2 = max\{lower - 1 + v_1, 0\}$ and $upper_2 = max\{upper - 1 + v_1, 0\}$. Consequently, $0 \leq lower_2 \leq 1, 0 \leq upper_2 \leq 1$, and the hypothesis is applicable to the subformula rooted in the second child. Thus, during the evaluation of this subformula the condition of the lemma is satisfied and also $0 \leq v_2 \leq 1$. Then, value $v = min\{1 - v_1 + v_2, 1\}$ is computed, which satisfies $0 \leq v \leq 1$. The proof of inheritance for Algorithm 5 is finished. Now, let us consider Algorithm 6. If the second child of the implication root is not a leaf or the first child is a leaf, Algorithm 6 executes the "else" branch in case of implication. This case is entirely the same as the case of the implication in Algorithm 5, and it was proven just some lines above. Now, let us consider the other possibility, i.e., the lines in which the two algorithms differ at implication nodes (when the first child is not a leaf, but the second is). In this case, the second child is evaluated first by parameters $lower_2 = 0$ and $upper_2 = upper$. By the induction hypothesis, it gets a value v_2 such that $0 \leq v_2 \leq 1$. Further, the first child is evaluated by parameters $lower_1 = min\{1 + v_2 - upper, 1\}$ and $upper_1 = min\{1 + v_2 - lower, 1\}$. Knowing $0 \leq lower \leq 1$, $0 \leq upper \leq 1$, $0 \leq v_2 \leq 1$, also $0 \leq lower_1 \leq 1$, $0 \leq upper_1 \leq 1$ are satisfied. The hypothesis can be applied for the first child, and thus, its evaluation satisfies the condition of the lemma and also its value satisfies $0 \leq v_1 \leq 1$. Then, by $v = min\{1 - v_1 + v_2, 1\}$, the assigned value to the root also satisfies $0 \leq v \leq 1$. This case has also been proven. The induction works also for Algorithm 6.

Lemma 2. Each time Algorithm 5 (Prune1) and Algorithm 6 (Prune2) is called recursively, if the condition $0 \leq lower \leq upper \leq 1$ is fulfilled, then this will also hold for the parameters of the subsequent recursive call(s).

Proof. The fact that $0 \leq lower, upper \leq 1$ is already proven in Lemma 1. Here we prove only the part $lower \leq upper$. The proof goes by induction. The first call in both of the algorithms, to evaluate the root, i.e., the main formula itself, goes by $lower = 0$, and $upper = 1$, for which the condition of the statement definitely holds.

Now, as an induction hypothesis, assume that $0 \leq lower \leq upper \leq 1$ holds at the actual node. Let us prove that similar condition holds for its children, if any.

- At the first possible recursive call, i.e., if the actual node is a negation node, in both algorithms, $lower_1 = 1 - upper$ and $upper_1 = 1 - lower$ are used. Thus, applying the hypothesis, $0 \leq lower_1 \leq upper_1 \leq 1$ also holds.

- In case the actual node is a conjunction or a disjunction node, the two algorithms have similar code (the renaming of the children does not affect the following arguments). At a conjunction node, the first evaluated child is called with $lower_1 = lower$ and $upper_1 = 1$. Trivially, $0 \leq lower_1 \leq upper_1 \leq 1$ holds by the hypothesis. Then, a value v_1 is assigned to this child. From Lemma 1 it is known that $0 \leq v_1 \leq 1$. The second child is called with values $lower_2 = min\{1 + lower - v_1, 1\}$ and $upper_2 = min\{1 + upper - v_1, 1\}$ which can also be written in the form $lower_2 = 1 + min\{lower - v_1, 0\}$ and $upper_2 = 1 + min\{upper - v_1, 0\}$ clearly showing that $lower \leq upper$ implies $lower_2 \leq upper_2$.

- At a disjunction node, the first child is evaluated with parameters, $lower_1 = 0$ and $upper_1 = upper$. Thus, $lower_1 \leq upper_1$ holds by the hypothesis (or by knowing from Lemma 1 that) $0 \leq upper$. Then, the second child is evaluated by parameters $lower_2 = max\{lower - v_1, 0\}$ and $upper_2 = max\{upper - v_1, 0\}$ and thus, the hypothesis $lower \leq upper$ implies $lower_2 \leq upper_2$.

- At implication nodes in the root, let us analyse, first Algorithm 5. The first child is evaluated with parameters $lower_1 = 1 - upper$ and $upper_1 = 1$. Applying $0 \leq upper$, $lower_1 \leq upper_1$ is proven. The second child is evaluated with $lower_2 = max\{lower - 1 + v_1, 0\}$ and $upper_2 = max\{upper - 1 + v_1, 0\}$. Consequently, by $lower \leq upper$, the condition $lower_2 \leq upper_2$ also holds.
  In Algorithm 6, exactly the same parameters are used in the "else" branch, thus the proof of this case is exactly the same as for Algorithm 5.
  In the other case, when the second child is evaluated first, the parameters $lower_2 = 0 and upper_2 = upper$ are used. Since $0 \leq upper$, the relation $lower_2 \leq upper_2$ is proven. Then, the first child is evaluated by $lower_1 = min\{1 + v_2 - upper, 1\}$ and $upper_1 = min\{1 + v_2 - lower, 1\}$ which can be written as $lower_1 = 1 + min\{v_2 - upper, 0\}$ and $upper_1 = 1 + min\{v_2 - lower, 0\}$. Clearly, $lower \leq upper$ implies
  $lower_1 \leq upper_1$ also in this case.

Thus, the condition is inherited for all the subsequent calls.

Lemma 3. For any expression having value x, if Algorithm 5 (Algorithm 6, resp.) assigns the correct value x to it, then one of the following statements is fulfilled.

a) The parameters have the relation $lower \leq x \leq upper$, and thus the assigned value is also between *lower* and *upper* (inclusively, allowing equality also).

b) If $x < lower$, then the algorithm assigns a value that is not larger than lower.

c) If $x > upper$, then the algorithm assigns a value that is not less than upper.

Proof. If the correct value is assigned to a formula, it always satisfies one of the three cases a), b) and c) depending on the values *lower* and *upper*: If the value of the formula x is between *lower* and *upper*, then case a) works. If x is less than *lower*, then case b) applies. Finally, if x is larger than *upper*, case c) is used.

Lemma 4. For any expression having value x, if Algorithm 5 (Algorithm 6, resp.) is called with parameters $lower = upper$, then it assigns

a) the value x of the expression correctly if $lower = x = upper$,

b) a value not larger than *lower* if the value of the formula x is less than *lower*, and

c) a value that is at least *upper* if the value of the formula x is greater than *upper*.

Proof. In this case, in both algorithms, we apply a cut, and the common value of $lower = upper$ is assigned to the formula in evaluation. Let the actual value of the formula be x. In case $x \neq lower$ (the formula has a value that differs from these bounds), then x is either less than *lower*, and the returned value is *lower*, or x is greater than *upper* and the returned value is the same as *upper*. Finally, if x has the same value as lower, then the correct value is assigned (case a). In either case, the statement of the lemma holds for any formula evaluated with parameters $lower = upper$. **Lemma 5.** For any expression having value x, if $lower < upper$, then Algorithm 5 assigns

a) the value x of the expression correctly if $lower \leq x \leq upper$,

b) a value not larger than *lower* if the value of the formula x is less than *lower*, and

c) a value that is at least *upper* if the value of the formula x is greater than *upper*.

Proof. The proof goes by induction on the height of the formula tree. The base of the induction, height 0, corresponds to the evaluation of every formula containing only a value and no connectives, that is, formula tree containing only a leaf node. Consequently, calling the algorithm to evaluate such formula, with parameters $lower < upper$ the "else" branch is executed by assigning the correct value of the leaf to v that is returned. Due to Lemma 3 the base case is proven. The induction hypothesis is as follows. Assume that the algorithm works correctly, that is, assigns the correct value of any expression having the height of its expression tree at most k (for any fixed nonnegative integer k) if the value x of the formula is between the values *lower* and *upper*, i.e., $lower \leq x \leq upper$. Moreover, it assigns a value at most lower if the formula has a lower value than *lower*; and it assigns a value at least *upper* if the formula has a greater value than upper. Let us prove the inheritance, let our formula have the height $k+1$. This part of the proof goes by cases (we call them Claims below) depending on the connective at the root of the formula tree. Notice that the condition for cut does not hold for the formula with height $k+1$ if its evaluation is called with $lower < upper$. (The case where an immediate cut is applied is already proven in Lemma 4.) **Claim 1.** Let the root of the formula having height $k+1$ represent a negation and let y be the value of the formula. If Algorithm 5 (Prune 1) is called with parameter values *lower* and *upper* such that $lower < upper$, then

a) Algorithm 5 assigns value $v = y$ to the formula if $lower \leq y \leq upper$; and

b) it assigns a value v to the formula with $v \leq lower$ if $y < lower$; and

c) it assigns a value v to the formula with $v \geq upper$ if $y > upper$.

Proof (of Claim 1). Let us see case a) first: when the formula has a value y such that $lower \leq y \leq upper$. The resulted value is computed by the formula $v = 1 - Prune1(child, 1-upper, 1-lower)$, that is, the algorithm is called with the child representing the main subformula and parameters $lower_1 = 1 - upper$ and $upper_1 = 1 - lower$. Notice that $0 \leq lower < upper \leq 1$ implies also $0 \leq lower_1 < upper_1 \leq 1$. The main sub-formula, in this case, has height k, and thus, by the induction hypothesis, we assume that its value x is correctly

computed by the algorithm if $1 - upper \leq x \leq 1 - lower$. This condition is fulfilled in our case since we have assumed that $lower \leq y \leq upper$, and consequently, the value $y = 1 - x$ is correct for our formula with negation in the root of its formula tree. If the value y of the formula is less than lower, then at the recursive call, its child is called with parameters $lower_1 = 1 - upper$ and $upper_1 = 1 - lower$. However, in this case, the value x of the child, $x = 1 - y$, is larger than $upper_1 = 1 - lower$. Thus, by the induction hypothesis the algorithm assigns a value v_1 to the child such that $v_1 \geq 1 - lower$. This implies that computing the value of the main formula, the assigned value v will be at most the value of $lower$. The inheritance in case b) is shown. Now, let us assume that the formula has a value y that is larger than $upper$. Then, its child is evaluated with $lower_1 = 1 - upper$ and $upper_1 = 1 - lower$. Since the value $x = 1 - y$ of the child is less than $lower_1 = 1 - upper$, by the induction hypothesis, the algorithm assigns a value v_1 to the child which is at most $1 - -upper$. Therefore, the value v assigned to our main formula will be at least upper.
Claim 1, i.e., the inheritance is proven for the case of negation. **Claim 2.** Let a conjunction be in the root of the formula having height $k + 1$, and let y be the value of this formula. If Algorithm 5 (Prune 1) is called with parameter values $lower$ and $upper$ such that $lower < upper$, then

a) Algorithm 5 assigns value $v = y$ to the formula if $lower \leq y \leq upper$; and

b) it assigns a value v to the formula with $v \leq lower$ if $y < lower$; and

c) it assigns a value v to the formula with $v \geq upper$ if $y > upper$.

Proof (of Claim 2). Evaluating our conjunctive expression having value y, first, the first child is evaluated by the call $v_1 = Prune1(child1, lower, 1)$, i.e., with $lower_1 = lower$ and $upper_1 = 1$. This, by the induction hypothesis (since this formula has height at most k), gives the correct value $v_1 = x$ of the formula rooted at the node child1 if $lower \leq v_1$. Otherwise, i.e., if $lower > x$, a value v_1 that is not larger than $lower$ is assigned.
Then, the algorithm for child2 is called by $v_2 = Prune1(child2, min\{1+lower-v_1, 1\}, min\{1+upper-v_1, 1\})$. There are three cases by the respective relations of the values of $lower$, v_1, and $upper$.

- If $v_1 \leq lower$ (which definitely happens in all cases when the value x of child1 is less than or equal to $lower$, then $min\{1 + lower - v_1, 1\} = 1$, and thus, $v_2 = Prune1(child2, 1, 1)$ which leads to a cut for the second child (independently of its real value z, $v_2 = 1$ is returned). Consequently, the value $v = v_1$ will be returned for the main conjunctive formula. On the other hand, if one of the children of a conjunctive formula has a value at most lower (as in this case the first child has value x), then it implies that the main formula itself cannot have a value larger than lower. Thus, knowing that both the real value y and the assigned value v of the main formula are less than or equal to $lower$, case b) satisfies and the inheritance for $v_1 \leq lower$ is proven.

- The second case occurs when $lower < v_1 < upper$. In this case, by the induction hypothesis the value of v_1 is correct, i.e., $v_1 = x$, it is exactly the value of the formula represented by child1. Then, $v_2 = Prune1(child2, 1+ lower - v_1, 1)$ is computed with parameters $lower_2 = 1 + lower - v_1$ and $upper_2 = 1$. Then, there are two cases based on the real value z of child2 and $lower_2 = 1 + lower - v_1$.
    - In case child2 has a value z such that $z \leq lower_2$, by the assumption of the induction hypothesis a value $v_2 \leq 1 + lower - v_1$ is computed and assigned. Thus, for the main formula of the conjunction gets value $v = max\{v_1 + v_2 - 1, 0\}$ is computed which has a value less than or equal to lower. On the other hand, if the value z of child2 is not more than $lower_2$ (i.e., $z \leq 1 + lower - v_1$), then the value y of our main formula cannot be more than $lower$, knowing that $v_1 = x$ correctly denotes the value of child1. Thus, case b) of the induction is proven, in case $z \leq 1 + lower - v_1$.
    - In the other case, child2 has a value z larger than $lower_2 = 1 + lower - v_1$. Then, by the induction hypothesis, the value $z = v_2$ of child2 is correctly computed. Consequently, the formula $v = max\{v_1 + v_2 - 1, 0\}$ uses both the correct values x of child1 and z of child2, and the value $v = y$ of the main formula is correctly assigned. Thus, case a) has been proven if $z > 1 + lower - v_1$.
- Now, let us see the third case, namely, when the value x of child1 is at least $upper$. Since child1 was evaluated by parameters $lower_1 = lower < upper$ and $upper_1 = 1$, and its value x satisfies $lower_1 \leq x \leq upper_1$, by the induction hypothesis, it gets exactly the value of the formula rooted at child1, i.e., $v_1 = x$. Thus, value v_1 has also the property $v_1 \geq upper$. The evaluation of child2 goes with $v_2 = Prune1(child2, 1 + lower - v_1, 1 + upper - v_1)$, i.e., with $lower_2 = 1 + lower - v_1$ and $upper_2 = 1 + upper - v1$. Now, there are three cases depending on the actual value z of child2.
    - The first case is when $z \leq lower_2 = 1 + lower - v_1$. In this case, by the induction hypothesis, v_2 is a value that is not more than $1+$

$lower - v_1$. Consequently, our main conjunctive formula is evaluated as $v = max\{v_1 + v_2 - 1, 0\}$. Thus a value v not more than lower is assigned to the main formula. We need to show that its real value y is also not more than lower. Actually, $y = max\{x + z - 1, 0\}$, using $v_1 = x$ and $z \leq 1 + lower - v_1$, we have $y \leq max\{x + 1 + lower - x - 1, 0\} = max\{lower, 0\}$. Thus, the conjunctive main formula cannot have a value y which is larger than $lower$. This first case has been proven.

  - The second case occurs if $1 + lower - v_1 < z < 1 + upper - v_1$. i.e., the value of child2 is between $lower_2$ and $upper_2$. In this case, by the hypothesis, the value z is the correct value of child2, that is, $v_2 = z$. Then, $v = max\{v_1 + v_2 - 1, 0\}$ is computed, where both $v_1 = x$ and $v_2 = z$ are exactly the values of the children, thus $v = y$ is also correctly assigned for the main formula.
  - Finally, in the third case $z \geq upper_2 = 1 + upper - v_1$. The value z of child2 and also the returned value v_2 are both at least $1 + upper - v_1$, by the hypothesis. Then, $v = max\{v_1 + v_2 - 1, 0\}$ is computed, where, actually, $v_1 + v_2 - 1 \geq upper$. Thus, the algorithm gives a value v that is at least $upper$. On the other hand, we need to show that the real value y also has this property. Knowing that $z \geq 1 + upper - v_1$ and $v_1 = x$, we have $y = max\{x + z - 1, 0\} = max\{v_1 + 1 + upper - v_1 - 1, 0\} \geq max\{upper, 0\} = upper$. The proof of this case, and so, the proof of Claim 2 is also finished.

Claim 3. Let a disjunction be in the root of the formula having height $k + 1$, and let y be the value of this formula. If Algorithm 5 (Prune 1) is called with parameter values $lower$ and $upper$ such that $lower < upper$, then

a) it assigns value $v = y$ to the formula if $lower \leq y \leq upper$

b) it assigns a value v to the formula with $v \leq lower$ if $y < lower$; and

c) it assigns a value v to the formula with $v \geq upper$ if $y > upper$.

Proof (of Claim 3). Child1 is evaluated by the algorithm with parameters $lower_1 = 0$ and $upper_2 = upper$. There are some cases listed below, based on the actual value x of the first child.

- Let us analyze the case $x < upper$. Then by the induction hypothesis, Algorithm 5 computed the correct value, i.e., $v_1 = x$ is returned. Then child2 is evaluated by $Prune1(child2, max\{lower - v_1, 0\}, upper - v_1)$, i.e., with parameters $lower_2 = max\{lower - v_1, 0\}$, and $upper_2 = upper - v_1$.

Now, there are two possibilities according to the fact if the condition $lower < x = v_1$ holds or not. Let us start with the case, when $x \leq lower$ also holds. In this case $Prune1(child2, lower - v_1, upper - v_1)$ is applied, i.e., $lower_2 = lower - v_1$. There are three possible subcases for the value z of child2 comparing it to the values $lower_2 = lower - v_1$ and $upper_2 = upper - v_1$. They are listed below.

- When $z < lower_2 = lower - v_1$, by the hypothesis, the returned value $v2 \leq lower - v_1$. Then computing $v = min\{v_1 + v_2, 1\}$, knowing that $v_1 + v_2 \leq lower$, a value v not more than *lower* is returned. On the other hand, let us see what we know about the real value y of the formula. Applying that $v_1 = x$ is the actual value of child1 and the actual value of child2 $z < lower - v_1$, obviously, the value y of our main formula is less than *lower*, therefore case b) is satisfied.
- The second subcase occurs when $lower - v_1 \leq z \leq upper - v_1$. In this case, by the hypothesis, the value of child2 is correctly computed, i.e., $v_2 = z$. Since both the values of child1 and child2 are correct, using the formula $v = min\{v_1 + v_2, 1\}$ the correct value y of our main disjunctive formula is assigned, this subcase is proven.
- In the third subcase, we have $z > upper - v_1$. In this case, by the hypothesis, the returned value v_2 is also at least $upper - v_1$. Substituting this condition into $v = min\{v_1 + v_2, 1\}$, we get that $v_1 + v_2 \geq upper$, and thus, a value v at least *upper* is assigned to our main formula. On the other hand, $z > upper - v_1$ and $x = v_1$ imply that the actual value y of our main formula is also larger than upper. Case c) is proven.

- Now let us consider the case when $lower < x < upper$. In this case $x = v_1$ still holds and child2 is evaluated by the call $Prune1(child2, 0, upper - v_1)$, i.e., by parameters $lower_2 = 0$ and $upper_2 = upper - v_1$. There are two subcases by the actual value z of child2.
  - First, let us see the case $z \leq upper_2 = upper - v_1$. Then child2 is correctly evaluated by the hypothesis, i.e., $v_2 = z$. Consequently, the value $v = min\{v_1 + v_2, 1\}$ is correctly assigned to the main formula. By Lemma 3, the case is proven.
  - Now, considering the case $z > upper_2 = upper - v_1$, by the hypothesis, the returned value v_2 is at least $upper - v_1$. Then, by $v = min\{v_1 + v_2, 1\}$, the return value for the main formula is computed, which has the value at least upper. Since the actual value of

child2, $z > upper - v_1$, where $v_1 = x$ is the actual value of child1, our formula has a value larger than *upper*. Thus case c) is proven.

- The last case for child1 occurs when $x \geq upper$. In this case, by the hypothesis, the computed value has also the value at least *upper*, that is, $v_1 \geq upper$. Then child2 is computed by the call $Prune1(child2, 0, 0)$, which leads to an immediate cut, having $v_2 = 0$. Then, the value of the main formula is computed by $v = min\{v_1, 1\}$ assigning $v_1 \geq upper$ to our formula. On the other hand, its actual value y cannot be less than *upper* since its first child has a value at least *upper*. Consequently, case c) occurs.

Claim 3, and thus, the inheritance for all cases with disjunction in the root is proven. The last possible case follows having implication as the main connective. **Claim 4.** Let an implication be in the root of the formula having height $k+1$, and let y be the value of this formula. If Algorithm 5 (Prune 1) is called with parameter values *lower* and *upper* such that $lower < upper$, then

a) it assigns value $v = y$ to the formula if $lower \leq y \leq upper$;

b) it assigns a value v to the formula with $v \leq lowerify < lower$; and

c) it assigns a value v to the formula with $v \geq upper$ if $y > upper$.

Proof (of Claim 4). Child1 is evaluated with the parameters, $lower_1 = 1 - upper$ and $upper = 1$, respectively. There are two cases, depending on the actual value x of child1: whether $x < lower_1 = 1 - upper$, or not.

- Let us start with the case $x < lower_1 = 1 - upper$. Then the returned value $v_1 \leq 1 - upper$, by the induction hypothesis. Further, child2 is evaluated by the parameters $lower_2 = 0$ and $upper_2 = 0$. This leads to an immediate cut with $v_2 = 0$. Then the return value is computed by $v = min\{1 - v_1 + v_2, 1\} = 1 - v_1$ that has value at least upper. On the other hand, we show a similar condition for the real value y of the main formula. Since the first main sub-formula of the implication has a value x less than $1 - upper$, the actual value y of the main implication formula cannot be less than *upper*, therefore case c) is applied and proven.

- Let us see the case when $x \geq 1 - upper$, then, by the hypothesis, $v_1 = x$ is correctly assigned. Then, there are two subcases based on the value of the condition $v_1 < 1 - lower$.
  - If this condition $v_1 < 1 - lower$ is true, then child2 is evaluated by the parameters $lower_2 = 0$ and $upper_2 = v_1 - (1 - upper)$. Consequently,

there are two possibilities based on the value z of child2. They are detailed below.

    - If $z \leq v_1 - (1 - \mathit{upper})$, then by the hypothesis, it is correctly computed and $v_2 = z$. Then, the value of the implication is correctly assigned based on the fact that the assigned values are correct for both children. By Lemma 3 the case is proven.
    - The other possibility for child2 is that $z > v_1 - (1 - \mathit{upper})$. In this case, by the hypothesis, also the assigned value $v_2 \geq v_1 - (1 - \mathit{upper})$ has this property. Then, by computing the value of the main formula, $v = \mathit{min}\{1 - v_1 + v_2, 1\} \geq \mathit{upper}$. On the other hand, we can establish the following about the real value y of the main implication formula. In this case, since $x = v_1$ and $z > v_1 - 1 + \mathit{upper}$, the actual value y of our main formula is more than *upper*, thus case c) is proven.

- Now, we are looking at the case, when $v_1 \geq 1 - \mathit{lower}$. In this case child2 is evaluated with parameters $\mathit{lower}_2 = v_1 - (1 - \mathit{lower})$ and $\mathit{upper}_2 = v_1 - (1 - \mathit{upper})$. There are three possibilities for the value z of child2.
    - The first is $z < \mathit{lower}_2 = v_1 - (1 - \mathit{lower})$. By the hypothesis, also $v_2 \leq v_1 - (1 - \mathit{lower})$. In this case, computing the value of the main formula, $v \leq \mathit{lower}$. Also, the actual value y is less than *lower*, thus case b) applies.
    - The second case happens, when $\mathit{lower}_2 = v_1 - (1 - \mathit{lower}) \leq z \leq \mathit{upper}_2 = v_1 - (1 - \mathit{upper})$. Then, by the hypothesis, the value $v_2 = z$ is correct, and then, the value v of the implication is correctly computed (both children have their correct values assigned). By Lemma 3, this case is proven.
    - In the third case, $z > v_1 - (1 - \mathit{upper}) = \mathit{upper}_2$. Then, by the hypothesis, the assigned value $v_2 \geq \mathit{upper}_2 = v_1 - (1 - \mathit{upper})$. Then, evaluating the main formula, we get $v \geq \mathit{upper}$, and also the actual value cannot be less than *upper* since the second child has a value larger than $v_1 - (1 - \mathit{upper})$, where $v_1 = x$ is the actual value of child1. Thus, case c) is proven for this possibility.

The proof of Claim 4 is finished.

The previous claims imply that the algorithm Prune1 works for all formula trees up to height $k+1$ satisfying the conditions of the lemma. Thus, the proof of the lemma by induction is completed, the statement holds for every formula with any length.

Lemma 6. For any expression having value x, if $lower < upper$, then Algorithm 6 assigns

a) the value x of the expression correctly if $lower \leq x \leq upper$,

b) a value not larger than lower if the value of the formula x is less than lower, and

c) a value that is at least upper if the value of the formula x is greater than upper.

Proof. Similar to the proof of Lemma 5, the proof goes by induction on the height of the formula tree. Since the two algorithms Prune1 (Algorithm 5) and Prune2 (Algorithm 6) share a large part of their code, in the present proof for the sake of simplicity we detail only the differences. The base case and the induction hypothesis are exactly the same for Prune2 as they are for Prune1. The inheritance can be proven by similar Claims based on the main connective of the complex formula having a tree with height $k+1$. Actually, at the case of negation in the root, the code of Prune2 is exactly the same as the code of Prune1, thus Claim 1 and its proof shows the inheritance. In the case of conjunction or disjunction at the root of the formula tree with height $k+1$, similar statements as Claim 2 and Claim 3 can be proven for Algorithm 6. The only difference between the codes of the Algorithm 5 and Algorithm 6 is that in case child1 is not a leaf node, but child2 is a leaf, the children are permuted. However, in Łukasiewicz logic both conjunction and disjunction have the commutative property, which proves that by interchanging the order of child1 and child2 the formula has the same value as without this change. Based on that, the inheritance of our inductive proof in cases of conjunction and disjunction in the root, for Algorithm 6 is also proven. The most interesting case is when the root contains an implication. Algorithm 6 has an additional condition with a new part of the code (comparing it to Algorithm 5). For this case we need to prove a statement similar to Claim 4. **Claim 4'.** Let an implication be in the root of the formula having height $k+1$, and let y be the value of this formula. If Algorithm 6 (Prune 2) is called with parameter values lower and upper such that $lower < upper$, then

a) it assigns value $v = y$ to the formula if $lower \leq y \leq upper$;

b) it assigns a value v to the formula with $v \leq lowerify < lower$; and

c) it assigns a value v to the formula with $v \geq upper$ if $y > upper$.

Proof (of Claim 4'). When child1 is a leaf node or child2 is not a leaf, Prune2 executes the same code as Prune1, thus the proof of Claim 4 in the proof of Lemma 5 suffice. However, if child1 is not a leaf node, but child2 is a leaf, Algorithm 6 executes a different code. Let us prove the inheritance for this case. If the "if" condition at implication node holds, then child2 is evaluated first, moreover child2 is a leaf. The case $upper = 0$ is not possible since it was already leading to a cut for this implication root as well (the case $lower = upper$ was already studied in Lemma 4). Since $upper > 0$, the leaf child2 is evaluated and gets its correct value, that is $v_2 = z$. Now, child1 is evaluated by Prune2 with parameters $lower_1 = min\{1 + v_2 - upper, 1\}$ and $upper_1 = min\{1 + v_2 - lower, 1\}$. There are three possibilities by the value v_2:

- The first possibility is $v_2 < lower$. Then, v_1 is evaluated by parameters $lower_1 = 1 + v_2 - upper$ and $upper_1 = 1 + v_2 - lower$. Now, there are three subcases depending on the actual value x of child1.

  - If $x < lower_1 = 1 + v_2 - upper$, then by the hypothesis, its assigned value $v_1 \leq 1 + v_2 - upper$. Then, the computed value for the root $v \geq upper$. On the other hand, the real value y of the main implication formula can also be analyzed. By knowing that $v_2 = z$ is the correct value of child2 and $x < 1 + v_2 - upper$, the actual value y of the implication must have a value less than the value of $upper$. Therefore, case c) applies and shows the inheritance.
  - The second subcase occurs if $lower_1 = 1+v_2-upper \leq x \leq upper_1 = 1 + v_2 - lower$. By the induction hypothesis, in this case $v_1 = x$ is correctly assigned. Since both children have their actual values correctly assigned, the result of the implication formula is also correctly assigned. By Lemma 3, the inheritance is clearly shown.
  - Let us consider the third subcase, when $upper_1 = 1+v_2-lower < x$. Then, by the hypothesis, the assigned value $v_1 \geq 1 + v_2 - lower$. Consequently, when computing v, $v \leq lower$ holds. On the other hand, $v_2 = z$ is the correct value of child2, and then, $1+v_2-lower < x$ for child1 implies that the actual value of the implication is less than $lower$. In this way, case b) is proven.

- Now, case $lower \leq v_2 \leq upper$ is studied. After evaluating child2 and having $v_2 = z$, $Prune2(child1, 1 + v_2 - upper, 1)$ is called, i.e., child1 is evaluated with parameters $lower_1 = 1 + v_2 - upper$ and $upper_1 = 1$. There are two subcases by the actual value x of child1.

  - If $x_1 \leq v_2 \leq upper$, then, by the hypothesis, the assigned value v_1 is not more than $1 + v_2 - upper$. Then, evaluating the implication formula we have $v = min\{1 - v_1 + v_2, 1\} \geq upper$. However, in this case, also the actual value y of the implication formula is $y = min\{1-x+z, 1\} = min\{1-x+v_2, 1\} \geq min\{1--(1+v_2-upper)+v_2, 1\} = upper$, i.e., y is also not less than $upper$. Case c) is shown.

  - In the other case, when $x \geq lower_1 = 1 + v_2 - upper$, its value is correctly assigned by the hypothesis, i.e., $v_1 = x$. Since both children have correct values assigned, also v is correctly computed. The inheritance is, then, follows by Lemma 3.

- The third case for child2 is $v_2 > upper$. In this case $Prune2(child1, 1, 1)$ is called (i.e., $lower_1 = upper_1 = 1$ leading to an immediate cut with returned value $v_1 = 1$. Then $v = min\{1 - 1 + v_2, 1\} = v_2$ is assigned that is a value larger than $upper$. On the other hand, knowing that child2 has a value $z = v_2$ larger than $upper$, the implication formula must also have a value larger than $upper$. Case c) is proven in this case.

The proof of Claim 4' is finished. This implies that Algorithm 5 also works for all formula trees up to height $k+1$ satisfying the conditions of the lemma. Thus, the proof by induction is completed, the statement holds for every formula with any length.