MANCHESTER 1824
The University of Manchester

Verifying Binarised Neural Networks using SMT-Based Model Checking

A dissertation submitted to The University of Manchester for the degree of
Master of Science in Artificial Intelligence
in the Faculty of Science and Engineering

Year of submission
2022

Student ID
10880805

School of Engineering

Contents

Word count: 15354

Abstract

Due to the broader range of use of AI, it is becoming essential to verify the reliability and robustness of the software. Most verifiers have tried to verify ANNs based on the SAT method but have difficulty dealing with the rapid increase in ANN size. Thus, we focus on Binarised Neural Network(BNN) for more accurate and faster ANN verification. BNN is a type of ANN that can binarise weights, biases, and activations at runtime with fewer floating point errors and lower computational complexity. We use ESBMC to verify that BNNs satisfy safety properties in a reasonable time. ESBMC is a C/C++ verification program based on SMT that supports various solvers. Three parts are needed to verify BNN using ESBMC: (i) BNN training, (ii) BNN testing (iii) BNN verification. We train and test BNNs for MNIST image classification and introduce multiple optimisation strategies for efficient and accurate verification. We verify the testing program through ESBMC to check the satisfiability of safety properties. The safety properties are defined by applying the L_∞ norm to MNIST to verify this test code in ESBMC. As a result, we demonstrate that ESBMC can verify multilayer BNNs with hundreds of neurons for each layer within a reasonable time.

Declaration of originality

I hereby confirm that this dissertation is my own original work unless referenced clearly to the contrary, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual property statement

i The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made *only* in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations (see `http://www.library.manchester.ac.uk/about/regulations/_files/Library-regulations.pdf`).

1 Introduction

1.1 Background and motivation

Artificial Intelligence (AI) is a technology to simulate human cognitive skills such as problem-solving and pattern recognition by utilising computers. There are two types of artificial intelligence: one is an expert system based on rules, and another is an artificial neural network technology based on machine learning. Artificial neural network (ANN) technology has recently gained popularity through machine learning compared to expert systems. Unlike expert systems, ANN does not require experts to create thousands of rules and generally shows higher accuracy in various fields.

ANN, the AI technology inspired by the structure of human brain neurons, imitates how neurons work artificially. Like human brains, ANN consists of multiple layers, each with multiple neurons. We can train ANNs with enormous-sized data and utilise the trained-ANNs for various purposes, such as classification, regression, and clustering. ANN technology is widely used in diverse fields using these capabilities, from AI assistants and self-driving cars for individuals to factory automation for industries. The use of ANN has spread in the IoT field, and now it is necessary to use ANN on embedded devices. However, traditional ANN techniques are problematic for embedded software because they require massive CPU and memory use.

Furthermore, the proliferation of ANN may pose significant risks to humans. Currently, ANN is vulnerable to advertising attacks due to its black box nature. An ANN consists of thousands of neurons with multiple layers that are extremely challenging to analyse, which results in difficulties in verification[1]. More specifically, ANN verification takes an enormous time and is less accurate due to floating point errors. This susceptibility can lead to severe threats like self-driving car accidents and personal information leakage from AI assistants. Thus, there is a strong need to ensure the safety and reliability of ANNs so that ANNs do not cause danger to humans.

Binarised Neural Network (BNN) was introduced to address these problems of ANN by binarising parameters such as weights and activations [2]. Binarisation can offer various advantages. First, BNNs can be efficiently performed even on low-performance devices, allowing AI's range of use to be extended to embedded devices [3]. Second, BNN also allows more accurate verification by minimising errors caused by floating point arithmetic when calculating gradients during backward propagation [2], [4].

We can use software verification technologies to guarantee safe and reliable BNNs. The consuming time of verifying ANNs generally takes massive time at the moment due to the complex structure of ANNs. Even worse, the current verification time for BNNs is considerably longer than the ANN verification time [4]. BNNs are also less scalable than ANNs for verification if the network size increases [4]. Moreover, BNNs, like other general ANN networks, adopt deep learning methods and more and more layers and neurons are required, demanding longer verification time to verify large-

sized BNNs for deep learning [4]. Therefore, there is a significant need for a method that can verify BNNs efficiently and accurately.

In this regard, we focus on verifying BNNs in a reasonable time for this study. Specifically, we verify the safety properties of pre-trained BNNs using ESBMC, an SMT-based model checking tool. We define safety properties by applying the L_∞ norm to the input and output data. And then, we verify that the BNNs are satisfied within the range of these safety properties. Furthermore, the verification time metrics are compared while increasing the number of layers and neurons to confirm whether verification is possible within a practical time range.

1.2 Aims and objectives

The primary purpose of this work is to verify BNNs using SMT model checking to ensure reliable BNN implementation in a reasonable verification time. For this goal, we propose methods to implement optimised BNN training and testing algorithms for ESMBC and verify it efficiently through ESBMC. We also conduct experiments to check the verification time metrics. In more detail, this study makes the following contributions.

1. We verify that the BNN models satisfy the safety properties with MNIST image data and ESBMC, an SMT-based verifier.
2. We propose optimisation strategies to reduce the verification time on ESBMC by using lookup tables, macro functions and conversion data type to boolean data type.
3. We conduct three sets of experiments, and each experiment consisted of three-layer BNNs, five-layer BNNs, and seven-layer BNNs. Three solvers, Boolector, Bitwuzla, and Yices, are used for each set, and BNN verification is performed while increasing the number of neurons in the hidden layer from 10 to 200.
4. We show that verifying medium-sized BNNs on ESBMC is possible in a reasonable time by adopting the proposed optimisation strategies. We also demonstrate that the Boolector solver offers the best scalability.

1.3 Report structure

This report consists of a total of four parts. The first part is the Introduction part, which introduces the background and reasons why we perform this research and explains the purposes of the study. The second part is the Literature review, which describes the existing studies and background knowledge required for this study. The third part is the Methods, which demonstrates how we implement the methods to train, test, and verify the BNNs. The fourth part is the Results and discussion,

which explains and compares the results of the experiments for the three-layer BNNs, five-layer BNNs, and seven-layer BNNs with different solvers. The last part is Conclusions and future work, which describes the achievements in this study and suggested improvements for further studies.

2 Literature review

2.1 Artificial Intelligence(AI)

As the name suggests, Artificial Intelligence(AI) is a technology that makes computers intelligent to learn and infer like humans. AI is primarily divided into two fields: symbolic AI, a rule-based strategy, and artificial neural networks(ANN) based on machine learning. Symbolic AI has the advantage of easier comprehension because understanding how rules work is straightforward to humans. However, symbolic AI requires many experts in each field to create numerous rules to teach computers. This method has brought severe limitations for a symbolic AI in that it is challenging for people to generate all existing rules correctly, and the number of experts is also limited. Due to these limitations, ANN has been widely used in recent years, and in general, when speaking of AI, we refer to ANN. ANN has recently shown significant performance improvements and has been applied to various fields, including natural language processing, computer vision and robotics. However, the black box nature of ANN has caused severe concern. Since ANN has a highly complex structure for people to understand, ensuring the safe use of ANN is tricky. In order to develop an understandable ANN, research on a neuro-symbolic AI method, a combination of symbolic AI and ANN, has also been studied recently.

Machine learning, a way of training ANN, comprises supervised, unsupervised, and reinforcement learning. Supervised learning is a method of teaching data with answers to computers. For example, tons of cat images are labelled as cats, and then computers are trained that these images are cats. Supervised learning can be used for regression and classification, as shown in the example above. On the contrary, unsupervised learning is a method of learning the learning data itself without teaching the correct answer. For a similar example, the computer learns the images of various people. The computer analyses and learns the pattern of each data. Then, the computer can cluster similar faces from the images. Therefore, unsupervised learning is appropriate for clustering and association problems. Finally, reinforcement learning is an agent-based method that learns computers by finding the optimal answer in a way that rewards the computer when it presents the correct answer. Reinforcement learning is suitable for learning various games such as AlphaGo.

Deep learning is a kind of machine learning and has recently been in the spotlight. Deep learning and machine learning differ in whether feature engineering for data is necessary. Machine learning extracts important features from raw data by humans. On the other hand, deep learning does not require feature engineering and can learn raw data directly. Deep learning is more appropriate when prior knowledge of data does not exist because it does not require feature engineering. Also,

Deep learning can handle enormous size data and usually perform well on this kind of data. However, there are several disadvantages of deep learning. Accuracy is likely low when the data size is not big enough to be suitable for deep learning. Also, deep learning requires high-performance computing resources and a longer training time than machine learning.

2.2 Artificial Neural Networks (ANNs)

2.2.1 Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs), a technology inspired by the human brain, mimic the structure and function of human neural networks using computers. Human neurons are connected in a multilayer structure to process signals. ANN takes the idea from the structure, implements artificial neurons, and connects them in a multilayer to process input data.

The ANN structure consists of an input layer, one or more hidden layers, and one output layer, and the layer can be either fully connected or partially connected. Figure 1 shows an example of the structure of a fully-connected multi-layered ANN. The input layer receives and processes the data and sends it to the hidden layers, which process it and send it to the output layer again. Finally, the output layer processes the final result value of the ANN.

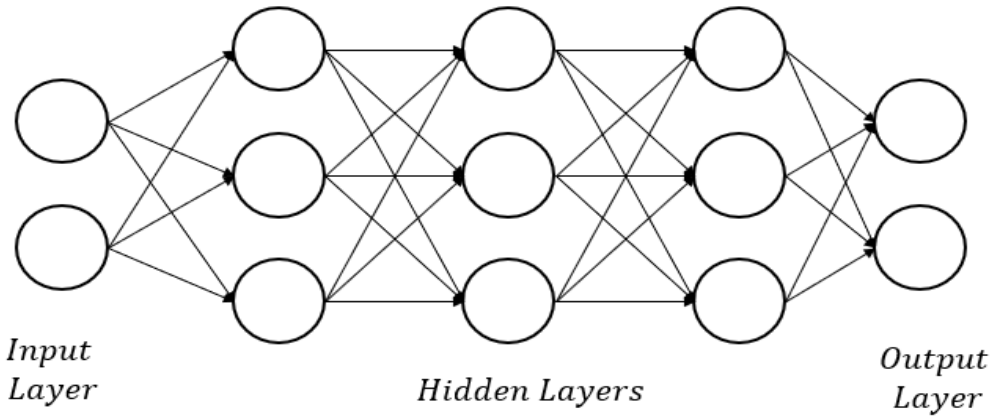

Fig. 1. The structure of a fully-connected multi-layered ANN

Figure 2 briefly illustrates the data processing structure of ANN. The ANN receives input data x. The ANN's weights w are multiplied by the input data x to obtain the summation and send this value to an activation function. An activation function determines whether to activate this value. An activation function also obtained an idea from biological neurons. Biological neurons receive input signals from other neurons and process them to send output signals, which only send signals to other neurons when the input signal exceeds a specific value. Similarly, ANN's activation function multiplies the input data by weights to receive the added bias and then determines how much

output signal is sent according to certain criteria. Since there are diverse activation functions, the standards for each activation function are different.

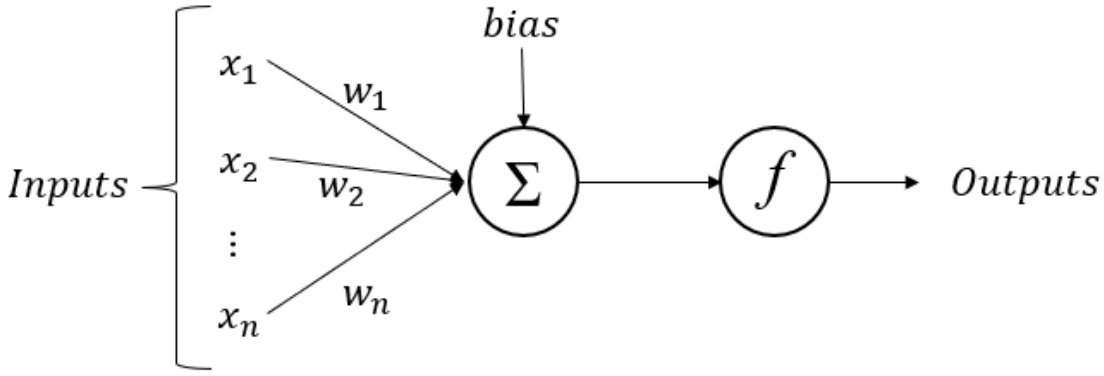

Fig. 2. The general model of an ANN

2.2.2 Forward Propagation

The basic propagation method is described in Figure 2.

1. Multiply the input data x by the weight w and add the $bias$.

$$sum(x, w, bias) = \sum_{i=1}^{n} x_i w_i + bias \tag{1}$$

2. Transfer this value as the input value of the activation function to obtain the output value.

$$activation\_function(sum(x, w, bias)) = output \tag{2}$$

3. If there is a connected layer, it transmits the output value to the layer. In the case of the last output layer of the ANN, the ANN determines the outcome.

The input data x value in the ANN is fixed, so it is crucial to derive the correct output value by finding the proper weights, biases, and activation functions. The variables, like activation functions, are called hyperparameters, and we can tune them. For instance, we can find the most suitable kinds of activation functions and the number of layers and neurons for ANNs. On the other hand, the variables like weights and biases are called parameters, and we cannot tune them. Parameters can only be learned from training. In order to discover the most optimised parameters for the given ANN, the backward propagation method was devised.

2.2.3 Backward Propagation

The backward propagation method is primarily used to discover each ANN's optimised weights and biases. Backward propagation uses the error between the predicted value by ANN through forward propagation and the real target value. Contrary to the forward propagation method of obtaining the output value from the input layer through the hidden layer, it is called backward propagation because it propagates errors from the output layer to the input layer.

There are several kinds of loss functions, which are methods of calculating errors in backward propagation.

1. Mean Squared Error(MSE): MSE is the most basic and straightforward loss function. The loss is calculated by squaring the difference between the predicted and actual values and dividing the value by the total number of data, N, for obtaining an average. The MSE method has the disadvantage that the error value obtained from MSE is larger than the actual error because the difference between the predicted value and the actual value is always squared to make it positive.

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(prediction_i - target_i)^2 \tag{3}$$

2. Root Mean Squared Error(RMSE): RMSE has been introduced to reduce errors in MSE. By obtaining the root value in MSE, the error size generated from the square is reduced.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(prediction_i - target_i)^2} \tag{4}$$

3. Cross entropy Loss: The Cross entropy method is also one of the most commonly used loss functions. It is a method of obtaining the difference between the actual and predicted values through the cross entropy method. If two classes need to be classified, binary cross-entropy is used. If there are more than two classes, geometric cross-entropy is used. t_i is the number of classes and p_i is probability for the ith class.

$$Cross\ entropy\ Loss = -\sum_{i=1}^{n} t_i \log(p_i)) \tag{5}$$

The loss gradient for all weights of the ANN can be obtained based on the chosen loss function among Cross entropy Loss, MSE, RMSE, et cetera [5]. We can update the weight by subtracting the loss gradient value multiplied by the learning rate from the previous weight value. The formula of weight update using the gradient of the loss function is as follows:

$$w\_updated = w - learning\_rate\ *\ \frac{dE}{dW}\,(E = loss\ function) \tag{6}$$

2.2.4 The Activation functions (ANNs)

The activation function receives the output value from the previous layer and determines how much this data is output according to a specific criterion. An essential feature of the activation function is nonlinearity because the errors in the real world are nonlinear[6]. In other words, ANNs can be trained more adequately using nonlinear activation functions [6]. There are various nonlinear activation functions such as sigmoid, hyperbolic tangent(tanh), and ReLU. In this study, the ReLU activity function is used among these.

(1) The Sigmoid function

The sigmoid function is a nonlinear activation function that was widely used in the early days of machine learning. The sigmoid function is suitable for binary classification problems because the result value is bisected into the range of x-values with values of 0 to 0.5 and x-values with values of 0.5 to 1, as shown in Figure 3. The formula and graph of the sigmoid and the derivative are as follows:

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{7}$$

$$Sigmoid'(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{8}$$

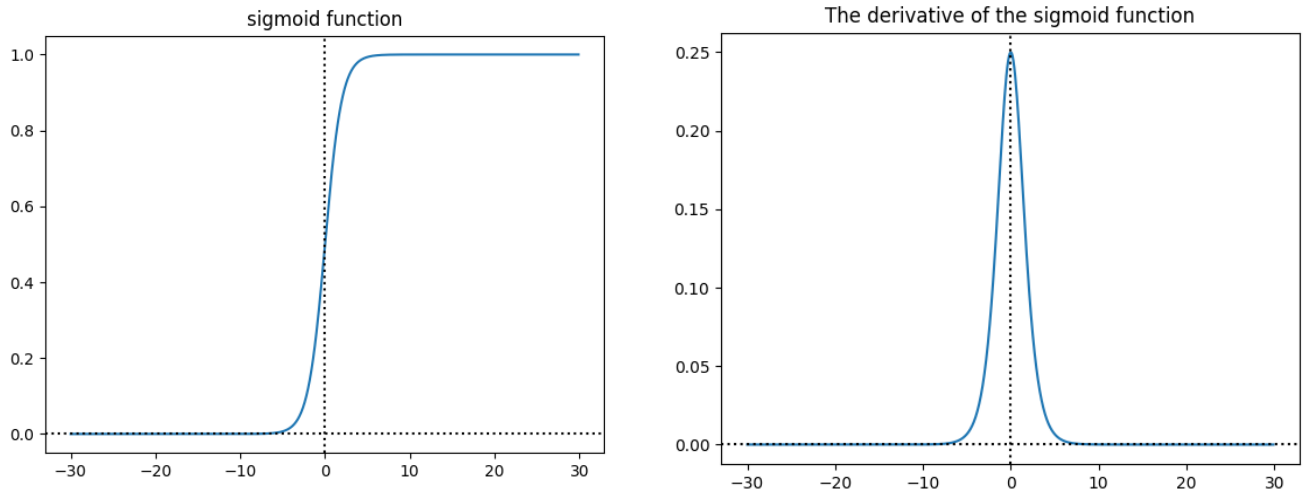

Fig. 3. The Sigmoid function and the derivative of the ReLU function

However, the sigmoid function can cause several serious issues, especially the gradient vanishing problem. In order to update parameters in machine learning backward propagation, a derivative of an activation function must be calculated. The gradient vanishing problem means that the gradients converge to zero if numerous differentiation is required during backward propagation with ANN with many layers. This convergence results in weights no longer being updated because it makes the gradient zero. As shown in Figure 3, the derivative of the sigmoid function converges to zero, and the gradient vanishes as the number of layers of ANNs grows. Also, it is more challenging to compute compared to other simple activation functions, such as the ReLU activation function.

(4) The Hyperbolic Tangent(tanh) function

Unlike the sigmoid activation function, in which the medium value of the function is 0.5, the hyperbolic tangent(tanh) activation function has a medium value of 0, as shown in Figure 4. A value range of -1 to 1 is also larger than the sigmoid with a range of 0 to 1. Due to this extended output range, the tanh function is more likely to converge faster and handle fault tolerance better than sigmoid [7]. The formula and graph of the tanh and the derivative are as follows:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{9}$$

$$tanh'(x) = 1 - (\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}})^{2} \tag{10}$$

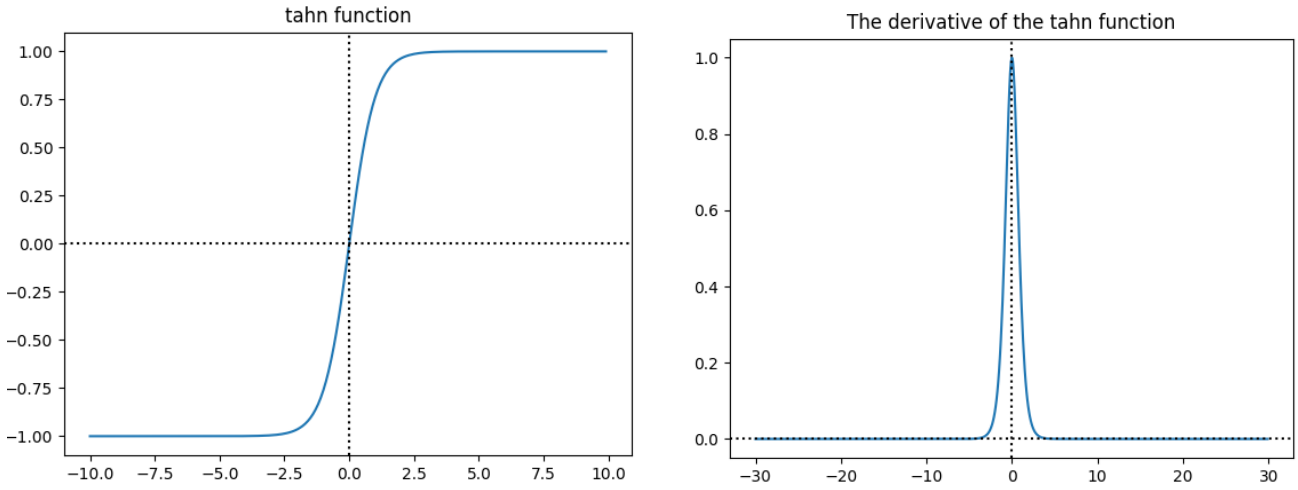

Fig. 4. The function of the tanh and the derivative of the tanh function

However, the tanh function still causes the gradient vanishing problem because the derivative of the tanh also converges to zero during backward propagation, as shown in Figure 4. In addition, the tanh operation is complex. Therefore, the hard tanh activation function, a simplified version of tanh, is occasionally used to speed up. The formula and graph of the hard tanh and the derivative are as follows:

$$Hardtanh(x) = \begin{cases} -1, & \text{if } x < -1. \\ x, & \text{if } -1 \leq x \leq 1. \\ 1, & \text{if } x > 1. \end{cases} \tag{11}$$

$$Hardtanh'(x) = \begin{cases} 0, & \text{if } x < -1. \\ 1, & \text{if } -1 \leq x \leq 1. \\ 0, & \text{if } x > 1. \end{cases} \tag{12}$$

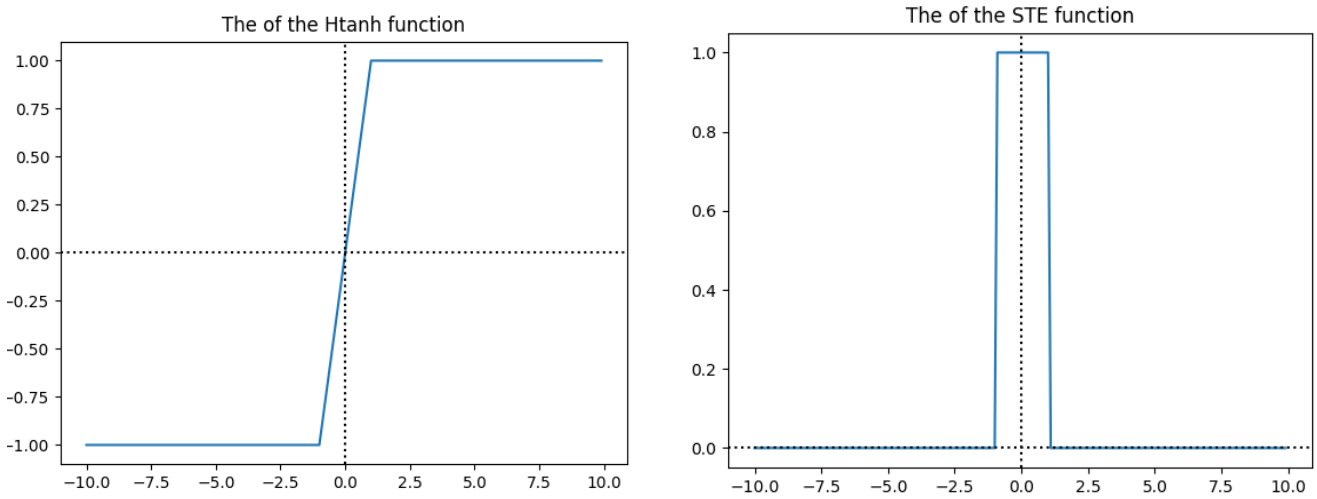

Fig. 5. The function of the hard tanh and the derivative of the hard tanh function

(3) The Rectified Linear Unit (ReLU) function

Recently, the most commonly used nonlinear activation function in machine learning is ReLU. The formula and graph of the ReLU and the derivative are as follows:

$$ReLU(x) = \begin{cases} x, & \text{if } x > 0. \\ 0, & \text{if } x < 0. \end{cases} \tag{13}$$

$$ReLU'(x) = \begin{cases} 1, & \text{if } x > 0. \\ 0, & \text{if } x < 0. \end{cases} \tag{14}$$

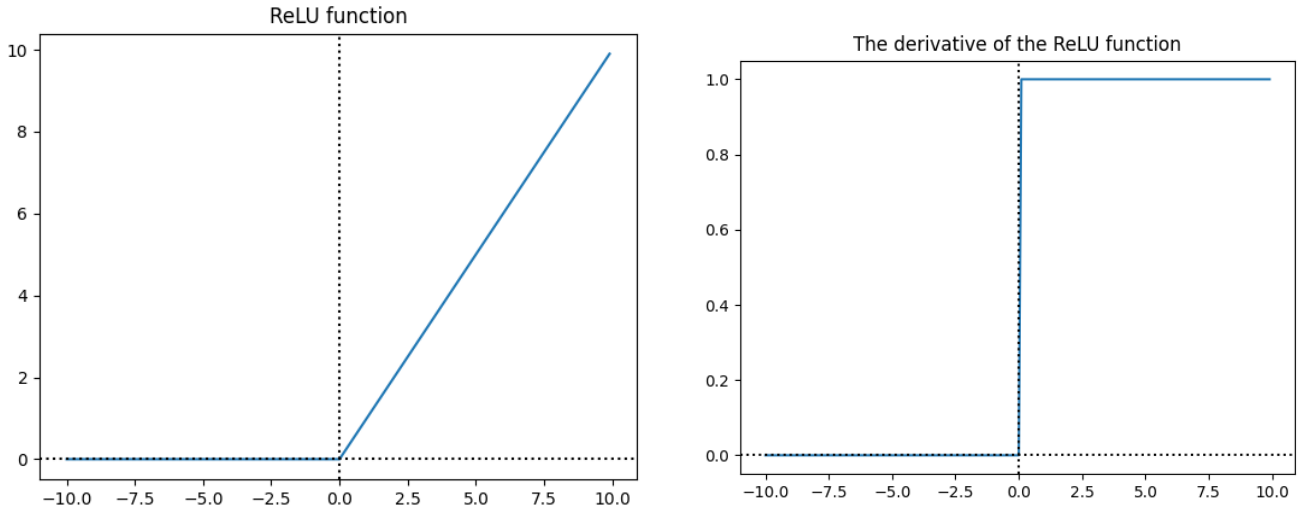

Fig. 6. The ReLU function and the derivative of the ReLU function

In this study, we use the ReLU function. ReLU has various advantages:

1. ReLU is straightforward to compute: The ReLU function is simple to calculate because it has

an uncomplicated formula. When an input value is greater than 0, the ReLU function returns an input value; otherwise, it returns 0. Unlike sigmoid and tanh activation functions, the ReLU function does not require intense computation, resulting in less resource use and faster arithmetic operation. [8].

2. ReLU does not cause gradient vanishing problem: As shown in Figure 6, when an input value is positive, the derivative of the ReLU function always returns 1, and when it is negative, it always returns 0. Therefore, the derivative of the ReLU function does not converge to zero, so it does not cause the gradient vanishing problem.

3. ReLU is easy to optimise: Deep learning networks based on ReLU activation functions are efficient to optimise because all values passed through the ReLU function are positive [9].

However, ReLU has a significant disadvantage. The ReLU function makes all negative values zero, so the weight update does not work well if the output value is negative.

(5) The Softmax function

The softmax is more frequently used in the output layer than the hidden layer since it has beneficial features for producing more accurate output but has a relatively complicated formula. In particular, the softmax function is suitable for multiple classification problems because it has the advantage that the number of output values is the same as the number of classes to be classified [6]. The formula of the softmax function is as follows:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}}, (n = the\ number\ of\ the\ classes) \tag{15}$$

2.2.5 Optimisers

We review the most widely-used optimisers to find appropriate weights and biases. Optimisers tune parameters to increase machine learning accuracy and help find optimal values for the parameters quickly and reliably. The most frequently used optimisers in recent years are as follows:

1. Gradient Descent (GD): The GD method finds the minimum value of the loss function by moving the slope of the loss function to the side where the slope of the loss function decreases based on an arbitrary starting point. The GD is an intuitive and straightforward way to find optimised parameters. However, the GD has several limitations. First, GD is slow because it calculates the gradients using all the data. In particular, recently used deep learning requires massive data; therefore, the gradient descent method is usually inappropriate for deep learning optimisation. Second, GD cannot guarantee that the global minimum can be obtained. If the random starting point is closer to the local minimum, GD likely finds the local minimum as the global minimum value.

2. Stochastic Gradient Descent (SGD): The SGD method was proposed to solve the problems of GD. Unlike GD, the SGD method uses randomly selected data samples. More specifically, SGD divides randomly sampled data into batch data of a specific size to find the most proper weights and biases, which results in faster optimisation than GD. Due to random sampling, the likelihood of finding local minima is also reduced. However, SGD also still has the probability of obtaining local minima as the global minima. In addition, SGD is usually more unstable than GD, with a risk of convergence failing.

3. Momentum: The Momentum method considers the value of the previous gradients when it chooses the data instead of randomly sampling it like SGD. The momentum algorithm accelerates a gradient if it continually points in a specific direction but controls it not to go in that direction if the gradient direction continues to change [10].

4. Adaptive Gradient(AdaGrad): The AdaGrad focuses on learning rates rather than directions like Momentum. The AdaGrad lowers the learning rate of variables with a high gradient and raises the learning rate of variables with a little gradient [11]. The AdaGrad method is called an adaptive gradient because it changes the learning rate according to the situation [11].

5. Adaptive Moments (Adam): The Adam method can be seen as a combination of Momentum and AdaGrad, and it is a method of finding optimised parameters by considering both the gradient direction and the learning rate.

2.2.6 Batch Normalisation

The SGD-based method trains ANN models by randomly sampling data from the original data and dividing the sampled data into batches of a specific size. Since each data is arbitrarily chosen, there is a high probability that each batch has a variety of distributions for each data. The batch normalisation algorithm normalises each data using the formula 16. Batch normalisation allows for higher accuracy and faster convergence [12]. The formula of the Batch Normalisation function is as follows:

$$Batch\_Normalisation(x) = \frac{x - E(x))}{\sqrt{Var(x) + \epsilon}} * \gamma + \beta \tag{16}$$

$E(x)$ means the average of the input value x, and $Var(x)$ means the variance of the input value x. In order to avoid division by zero, epsilon ϵ is added. The division of these two values is multiplied by gamma γ and then added beta β to complete normalisation. The gamma γ and beta β parameters are learnable during the training period.

2.3 Binarised Neural Networks (BNNs)

As the scope of applications for artificial intelligence has extended, there is now a need to employ machine learning, even in an embedded context. In particular, when machine learning has to be

applied to IoT domains, edge devices usually consist of low-performance hardware with limited computing resources, such as CCTV or street lights. The devices used in IoT commonly do not have enough CPU and memory to perform advanced ANN methods because they usually demand high-performance GPUs, CPUs, and massive memory use [3]. In order to apply machine learning to edge devices with limited computing performance, the BNN algorithm has been proposed.

The BNN algorithm is a type of ANN that uses binarised weights, and activations for runtime, which can drastically reduce memory and CPU usage during forward propagation [2]. This binarisation step enables BNNs to efficiently utilise machine learning even in low-power, low-memory environments such as IoT edge devices. Generally, a layer of a BNN consists of a total of three steps: binarisation, batch normalisation, and activation function. Deep learning through BNN is also possible by connecting multiple BNN layers.

2.3.1 Forward Propagation

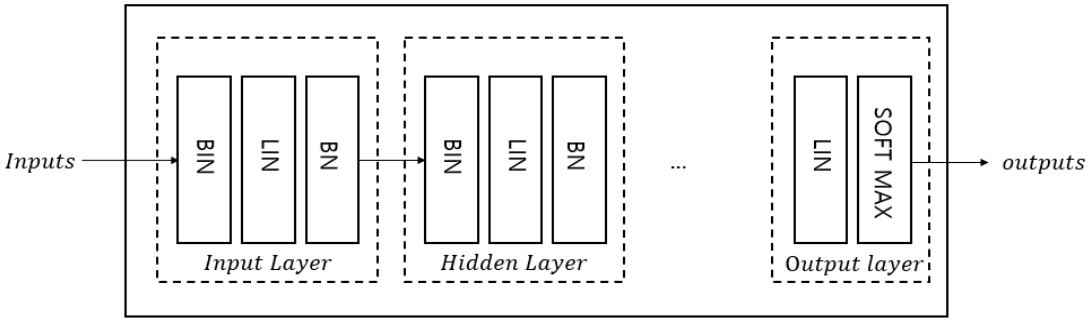

Fig. 7. An example structure of the BNN

BNN forward propagation also works by processing data in the order of the input layer, hidden layers, and output layer. Figure 7 shows the general structure of the BNN. The BNN in figure 7 consists of one input layer, several hidden layers, and one output layer, and all layers are fully connected. In the example of Figure 7, the structure of the input layer and the hidden layer is the same, and it is composed of a BIN layer, a LIN layer, and a BN layer. The BIN layer is a binarisation layer that binarises the input value. Next, the LIN layer is a linear transformation layer that multiplies the weight and the input value. Next, the BN layer is a batch normalisation layer that normalises each batch of data.

In contrast, the output layer comprises a LIN layer and a SOFTMAX layer. The LIN layer performs linear transformation in the same manner as other layers. Finally, the SOFTMAX layer uses the softmax function as an activation function, which yields output values by the number of classes it classifies. The index with the largest value among these output values is the predicted value of this BNN, and a loss can be obtained by comparing this value with the actual target value. Such a BNN

structure is an example, and BNN conversion is possible in various ways, such as adding or reordering layers.

(1) The binarisation layer

The binarisation layer binarises the received input value. It usually binarises all values to -1,1 or 0,1. Courbariaux et al. suggested two different binarisation algorithms: the deterministic method and the stochastic method [2].

The deterministic method is the easiest way to binarise the input values. This method outputs 1 if the input value is greater than 0 and outputs -1 if the input value is less than 0. If the deterministic binarisation of the -1,1 method is selected, the following formula is used:

$$Sign(x) = \begin{cases} 1, & \text{if } x >= 0. \\ -1, & \text{if } x < 0. \end{cases} \tag{17}$$

If the deterministic binarisation of the 0,1 method is selected, the following formula is used:

$$Sign(x) = \begin{cases} 1, & \text{if } x >= 0. \\ 0, & \text{if } x < 0. \end{cases} \tag{18}$$

The stochastic method is a method of binarising input values using probabilities and has the merit of having higher accuracy than the deterministic method, but it requires a longer calculation time.

The following formula is used if the stochastic binarisation of the -1,1 method is chosen.

$$Sign(x) = \begin{cases} 1, & \text{probability = hard\_sigmoid(x).} \\ -1, & \text{probability = 1-probability.} \end{cases} \tag{19}$$

The following formula is used if the stochastic binarisation of the 0,1 method is used.

$$Sign(x) = \begin{cases} 1, & \text{probability = hard\_sigmoid(x).} \\ 0, & \text{probability = 1-probability.} \end{cases} \tag{20}$$

This study aims to verify BNNs based on the C language in a reasonable time. In other words, we need to verify BNNs as fast as possible. Therefore, we selected the 0,1 binarisation method to limit the data type to the boolean type and chose the deterministic-based binarisation method to speed up the computation.

(2) The linear/affine transform layer

An affine transformation is a combination of linear transformation and translations. When we use

linear transformation, the n-dimension vector can be mapped to the m-dimension vector:

$$x \in \mathbb{R}^n \mapsto y \in \mathbb{R}^m \tag{21}$$

We can change the vector dimension by multiplying the activation and weight vectors. This expression can be represented as:

$$Linear\_Transformation(x) = \sum_{i=1}^{n}(activation_i \times weight_i) \tag{22}$$

By adding a bias to the linear transformation equation, we can make the affine transformation equation:

$$Affine\_Transformation(x) = \sum_{i=1}^{n}(activation_i \times weight_i) + bias \tag{23}$$

The linear transformation used in BNNs is the same as the affine transformation method used in ANNs. However, there is a difference in that activation, weights, and bias can be binarised.

(3) The batch normalisation layer

Batch normalisation used in BNNs is the same as ANN, a layer that normalises the data used in each batch. The only difference is that BNN uses binary weights and activations.

$$Batch\_Normalisation(x) = \frac{x - E(x))}{\sqrt{Var(x) + \epsilon}} * \gamma + \beta \tag{24}$$

The average of the input value x is denoted by $E(x)$, and the variance of the input value x is denoted by $Var(x)$. Epsilon ϵ is inserted to avoid division by zero. We divide these two numbers by gamma γ and multiply by beta β to normalise the data. The gamma γ and beta β parameters can be learned during the training phase.

(4) The activation function layer

The role of the activation function layer of BNN is also the same as that of ANN, and various activation functions such as sigmoid, tanh, and softmax are also available. The activation function of BNN also receives the preceding layer's output value and determines how much of it is output based on a given requirement.

$$activation\_function(x) = output \tag{25}$$

2.3.2 Backward Propagation

The method of training BNNs using backward propagation is similar to ANNs. BNN also defines the loss function and updates the weights and biases utilising optimisers such as SGD or Adam during the training phase. However, BNN performs binarisation using the sign function, but almost all values of the derivative of the sign function are zero [13]. Hence, the derivative of the sign function cannot be used as it is because it causes the gradient vanishing problem [13].

Courbariaux et al. solved this problem using the straight-through-estimator (STE) used in the Hinton (2012) study [13], [14]. In order to derive gradients of BNNs for backward propagation, the STE function is utilised instead of the derivative of the sign function [13]. Furthermore, because the STE function is used as the derivative, the activation function in backward propagation is the Hardtanh function rather than the sign function [13]. The formula and graph of graph and Htanh and STE functions are as follows.

$$Hardtanh(x) = \begin{cases} -1, & \text{if } x < -1. \\ x, & \text{if } -1 \leq x \leq 1. \\ 1, & \text{if } x > 1. \end{cases} \tag{26}$$

$$STE(x) = \begin{cases} 0, & \text{if } x < -1. \\ 1, & \text{if } -1 \leq x \leq 1. \\ 0, & \text{if } x > 1. \end{cases} \tag{27}$$

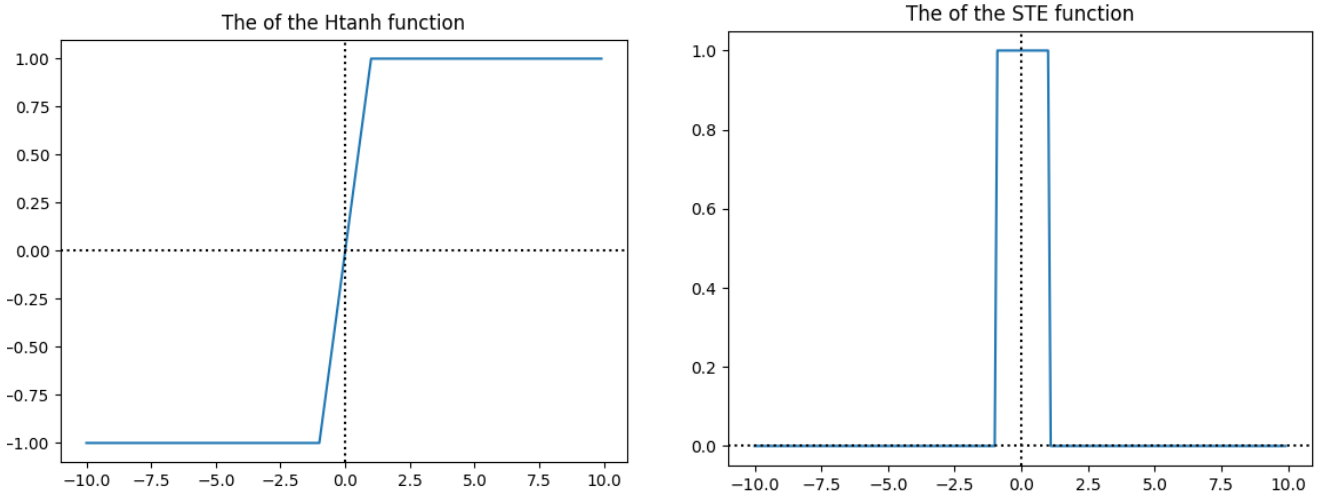

Fig. 8. The Hardtanh function and the STE function

2.4 Bounded Model Checking: SAT and SMT

Software verification is essential to implement secure and reliable software[15]. Bounded Model Checking (BMC) is a widely used software verification method. To verify the model using this method, we can simplify the model to a finite state machine and set safety or survival properties[16]. And then, BMC can verify that the model meets these properties and produces counterexamples if the model does not satisfy the properties [16]. Rather than verifying that model itself is safe, this approach is intended to prove that it does not contain errors in certain bounds [16]. The most frequently used BMC techniques include SAT and SMT techniques.

2.4.1 SAT (Boolean satisfiability problem)

The SAT method can determine the satisfiability of the software based on boolean formulas. In order to check satisfiability, SAT evaluates whether an interpretation exists that fulfils a specified boolean formula [17]. The software of model software can be proven safe if the counterexample is not discovered within a finite loop bound to the model through SAT verification. If the SAT method finds a counterexample, the verification is a failure, meaning the software contains errors; therefore, it does not guarantee safe execution.

The SAT model can be expressed by conjunctive normal form(CNF). Since the SAT model only uses boolean algebra, the operator only contains AND $\wedge$, OR $\vee$, NOT $\neg$ and parentheses. For example, a simple CNF of the SAT model is as follows:

$$(A \vee B) \wedge (\neg C) \tag{28}$$

The possible result values in the CNF above are shown in table 1.

Table 1. The results of the CNF

A	B	C	Result
TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE

As shown in Table 1, in the case of A=TRUE, B=TRUE, C=FALSE, the CNF result is true, but in the case of A=TRUE, B=TRUE, and C=TRUE, the CNF result is false. The SAT model is verified in this way.

As the SAT method consists of only boolean variables, this method has a limitation in that it cannot process formulas that have other more complicated variables, such as integers and real numbers. Even though AI-related applications, like other software, have been verified using the SAT method successfully, verifying AI programs that require more expressive logic with SAT alone is difficult [18]. For this reason, there is a strong need for a method to manage various variables with diverse operators.

2.4.2 SMT (Satisfiability modulo theories)

The SMT method was introduced to address the constraints of the SAT method by expanding the SAT method from using solely boolean logic to using first-order logic [1]. The first-order logic obtains ideas from the natural language we usually use and has a more comprehensive range of expressions than the boolean logic, which can only judge true and false. SMT can check the satisfiability based on first-order theories, a collection of first-order sentences. Consequently, SMT can support various data types: integers, real numbers, bit vectors, arithmetic, arrays, and recursive data types. SMT also offers a variety of symbols, and there are two types of symbols:

- Logical Symbols: Quantifiers(universal $\forall$, existential $\exists$), Logical connectives(AND $\wedge$, OR $\vee$, NOT $\neg$, implication $\rightarrow$, biconditional $\leftrightarrow$), parentheses, variables
- Parameters: Predicate symbols, Constant symbols, Function symbols, Equality symbols

Using these various symbols, SMT can verify more expressive logic like the following formula, which means that for every x and y, P(x) and Q(y) imply R(x):

$$\forall x \forall y \, P(x) \wedge Q(y) \rightarrow R(x) \tag{29}$$

In order to interpret SMT models, a combination of background theories is essential [1]. The first-order logic only satisfies the background theory if there is an assignment that can satisfy the union of background theory and first-order logic [1]. Various kinds of solvers based on the SMT method have been implemented, such as Z3, Boolector, Yices, and CVC4 et. Cetra

2.5 Safety properties for ANNs and BNNs

Safety properties refer to a collection of states the system must safely achieve during execution, which implies errors do not happen while executing the program [1]. Verifiers based on SMT can check whether the programs satisfy safety properties and provide counterexamples if they do not. Generally, we can define safety properties based on prior knowledge of data; however, because of

the black box nature of machine learning, the data we can use for setting the safety properties of the machine learning model is usually restricted to input data, and output data [1], [19], [20].

Since the input data used for machine learning is usually in the form of matrices consisting of numbers, it is possible to specify that the input data values are limited to a particular range. For classification problems, there exist correct answers called labels. The safety properties for output data can be defined to compare the predicted value with the actual value. If the predicted value is the same as the actual value, the safety property is satisfied, and if they have different values, the safety property is dissatisfied.

The range of the input data to set safety properties can be determined using L_1, L_2, and L_∞ norm. In the two-dimensional data, the formula of each norm is as follows.

1. L_1 norm:
$$L_1 = \sum |x_1 - x_2| \tag{30}$$
2. L_2 norm:
$$L_2 = \sqrt{\sum (x_1 - x_2)^2} \tag{31}$$
3. L_∞ norm:
$$L_\infty = max(|x|) \tag{32}$$

L_1 norm is simply a method of obtaining an absolute value for the difference between the two values. In contrast, the L_2 norm is a method of obtaining the square of the difference between the two values and then obtaining the square root of this value. Finally, the L_∞ norm, the most widely used method in machine learning, obtains the maximum data value. We also use L_∞ norm in this study.

2.6 ESBMC

ESBMC (Efficient SMT-based Context-Bounded Model Checker) is a bound model checker based on the SMT method, which supports the C/C++ language. In this study, we develop BNN test files in the C language and verify them on ESBMC. The primary purpose of ESBMC is to verify programs by checking whether a given file does not have errors. ESBMC provides diverse options to check the parse tree, symbol table, SSA, and SMT formula/model. ESBMC offers various solver options: Boolector, Z3, MathSat, CVC4, Yices, and Bitwuzla. The ESBMC supports the following features:

- Single/multi threaded C/C++ programs
- User specified assertions

- k-induction scheme
- Programming error detection: illegal pointer deferences, Out of bounds array access, Memory leak, Deadlock, et Cetra.

Figure 9 shows the structure of the ESBMC.

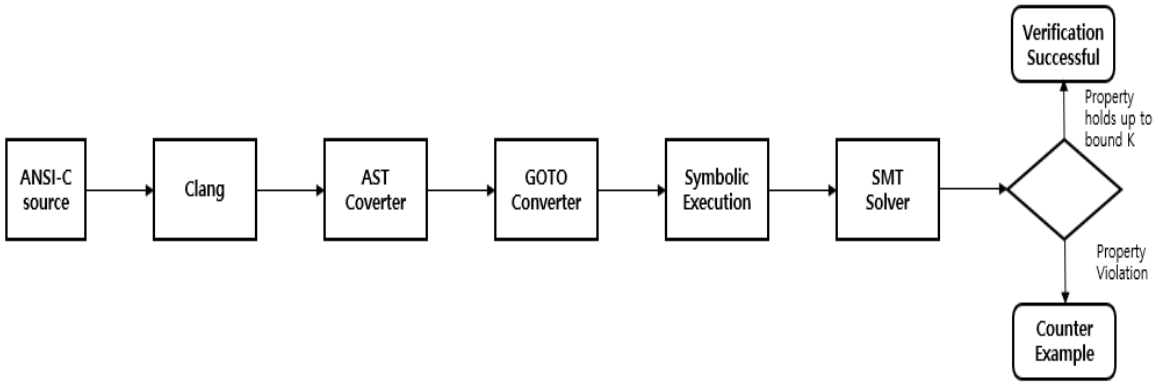

Fig. 9. A structure of ESBMC(Gadelha, 2018, p889)

1. Clang / AST Converter: The given ANSI-C code is converted to clang AST and then finally converted to ESBMC AST [21].
2. GOTO Converter: A GOTO program is generated to represent the state transition system [21].
3. Symbolic Execution: To create a static single assignment (SSA), ESBMC unwinds the generated GOTO program and then adds property checks and assertions [21]. ESBMC performs optimisation in this process for performance [21].
4. SMT Solver: Each assignment is encoded using the SSA set created in the previous step, and then the satisfiability is verified [21]. At this stage, the chosen SMT solver checks whether the program satisfies the given property, and at this time, various solvers such as Boolector, Z3, and Yices can be selected [21].

2.6.1 SMT Solvers

Depending on the goals of each program, there exist features that some solvers do not support. Since each solver focuses on different fields, it is essential to select an appropriate solver.

1. Boolector: focuses on the efficient management of bit-vector and arrays without quantifiers [22]. More appropriate reasoning is achievable due to this feature since it enables Boolector to create concrete models for arrays and bit-vector [22]. Since the data used for machine

learning is usually in the form of a multidimensional array, Boolctor is likely suitable for machine learning. Boolector also shows the best performance on stability and scalability in the experiment conducted in this study.

2. Yices: is a solver developed by SRI International that supports complex types such as predictive subtypes, real value, bit vectors, and boolean[23]. In particular, the Yices 2 version has improved performance and flexibility compared to the first version and also supports SMT-LIB 2.0 notation [23].

3. Z3: is a solver developed by Microsoft Research that combines satellite solvers that support array and arithmetics and E-matching abstract machines that support quantifiers based on DPLL methods and a core theory solver that manages equalities and uninterpreted functions [24].

4. MathSAT is a solver implemented by FBK-IRST and the University of Trento, which provides various functions such as incremental support, array, floating point, and SAT core[25].

5. CVC4 is a project implemented by the Cooperating Validity Checker 4 (CVC4) NYU and U Iowa, an extension of CVC [26]. It performs considerably more efficient for rich decidable logic, which supports features and SMT-LIBv2 features supported by existing CVCs[26], [27].

2.7 Existing SAT and SMT approaches for verifying BNNs

We review the existing method of verifying BNN using SAT and SMT solver. Narodytska et al. replaced BNN with the correct boolean formula and verified it using the existing SAT solver[28]. They measured the robustness of the BNN by setting the critical property [28]. However, the BNN size verified in this study is limited to medium-sized BNNs with between 100 and 200 neurons in each layer[28]. Andy Shih et al. also devised a verification method based on the SAT solver. In particular, they focused on cases when the input value was limited to a specific value [29]. They suggested an Algin-style learning algorithm focusing on the substitution of a neural network with an ordered binary decision diagram (OBDD) within a given input range and used an SAT solver to perform more efficient verification [29]. Jia et al. proposed an EEV system based on SAT solver for adequate and accurate BNN verification [4]. They introduced reified cardinality constraints to BNN encoding to improve an SAT solver speed, and a method to train BNNs suitably to verify in solver[4].

Furthermore, a verification method based on an SMT solver has been proposed. An SMT-based approach for verifying BNN suggested by Amir et al. is meaningful in that it is possible to verify not only binarised weights and activations but also non-binarised parameters [30]. Furthermore, various optimisation techniques such as verification query parallelisation were introduced to improve performance significantly [30]. Kovásznaia et al. have designed a portfolio solver that can utilise SAT, SMT, and MIP solvers to verify BNNs [31]. The portfolio solver can use all the SAT, SMT,

and MIP solver-based properties, which can run in parallel. This study also focuses on verifying medium-size BNNs [31].

2.8 Summary

The literature review consists of two essential parts: One is AI technology, a fundamental technology of BNNs, and another is the Bounded model checking methods for the verification. First, We describe AI, ANN, and BNN definitions and structures in detail. In particular, we focus on training methods: forward and backward propagation. Next, we explain two primary components of the Bounded model checking method: the SAT and SMT. We also illustrate the structure of ESBMC, the chosen SMT solver for this study, and the features of diverse solvers supported by ESBMC. Finally, we review the current methods of verifying BNNs based on the SAT or SMT.

3 Methods

3.1 Introduction

The primary purpose of this work is to verify BNNs to guarantee safety using ESBMC. Another important aspect is that the total elapsed time for verifying the BNN network has to be reasonable.

We have two main research questions.

1. Is it possible to verify pre-trained BNNs through ESBMC by asserting safety properties?
2. Is it possible to complete the verification in a reasonable time by employing various optimisation techniques?

In order to achieve these two goals, we conduct experiments based on empirical methodology. We develop training and test codes for BNNs and assert safety properties on the test code and make ESBMC verify them. In addition, we adopt diverse optimisation strategies to reduce the verification time. These methods include binarisation, Conversion data type to boolean data type, lookup tables and macro functions. We introduce the optimisation methods in detail for the testing section.

This study requires five steps to verify BNNs and measure the verification time.

1. Train BNNs to obtain pre-trained weights and biases.
2. Import the pre-trained BNNs and develop a forward propagation test code.

3. Assert safety properties based on the input and output data to check if the BNNs meet safety properties.

4. Verify the BNNs through ESBMC with different solver options: Boolector, Bitwuzla, and Yices.

5. Measure the number of assignments, maximum memory usage and verification time metrics, including total verification time, encoding to solver time, and runtime decision procedure.

3.2 Training

In order to generate pre-trained BNNs to get tested, a training period is necessary. We implement the training code in Python with the MNIST data set, which is 60,000 hand-written number images from 0 to 9. We train the BNNs by increasing the layer size to 3,5 and 7. The input layer size is consistently 784 because the image size of the MNIST image is 28*28. The number of neurons in the hidden layers increases from 10 to 200.

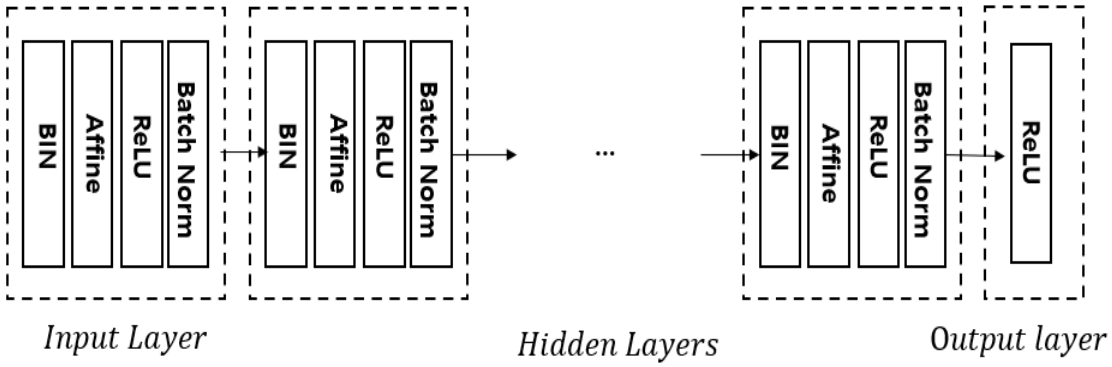

Fig. 10. The implemented structure of the BNN

The forward propagation proceeds in the order of the input layer, hidden layers, and output layer. Each input and hidden layer requires binarisation, affine transformation, batch normalisation, and a ReLU activation function. In contrast, the output layer consists only of a ReLU activation function. Algorithm 1 illustrates the entire process of the training period.

The backward propagation method proceeds in reverse to the forward propagation direction. The loss function to obtain the gradient was a cross-entropy loss function. The Adam optimiser was chosen for parameter optimisation since it can lessen the influence of scaling weights [13], [32].

3.2.1 The input data

We use the MNIST dataset to train the BNNs. The MNIST data consists of hand-written number images. The size of the image is $28 * 28$. There are 60,000 training images and 10,000 testing images. Therefore, we set the size of the input layer as $28 * 28(784)$.

Algorithm 1 The algorithm of the BNN training phase

```
1: procedure SET THE BNN MODEL
2:     for Every layer in the input layer and hidden layers do
3:         add binarise layer
4:         add affine transformation layer
5:         add ReLU activation layer
6:         add batch normalisation layer
7:     end for
8:     add ReLU activation layer for the output layer
9: end procedure
10:
11: set the BNN model
12: set the optimiser and loss function
13: for Every epoch do
14:     for Every batch do
15:         load the MNIST batch data
16:         do forward propagation
17:         get cross entropy loss
18:         do backward propagation to update weights, biases
19:     end for
20: end for
```

3.2.2 The Batch and Epoch

The default number of epochs is 100. The default size of the batch for the training is 64.

3.2.3 The Binarisation function

The binarisation layer transforms input data into binarised output values. In this paper, we implemented the binarisation algorithm deterministically for quick calculations and verification [2]. We binarise the data as 0 and 1 rather than -1 and 1 to limit the binarised values to boolean values because we use C language for testing. The mathematical expression of the deterministic binarisation function is:

$$Binarisation(x) = \begin{cases} 1, & \text{if } x >= 0. \\ 0, & \text{if } x < 0. \end{cases} \tag{33}$$

The determnisitic binarisation function returns 1, if the input is bigger than 0. Otherwise, it returns 0. As a result, the output values are restricted to 0 or 1, which can be considered boolean values in C.

1. The binarisation layer converts input into binary values: 0 and 1.
2. The deterministic binarisation function is adopted to change the float-valued or integer-valued data into binary data.

3. The weights, biases, and input data are binarised based on the deterministic binarisation function.

We use the sign function provided by Python. However, the Sign function provided by Python returns -1 if the value is negative and 1 if positive. For this study, the binarisation function should return 0 if the value is negative and 1 if positive. The value of Python's Sign function is multiplied by 1 and divided by 2 to convert the -1 , 1 binarisation scheme to the 0 , 1 binarisation scheme.

$$bin\_values = tensor.sign().add(1).div(2) \tag{34}$$

3.2.4 The Affine Transformation layer

The affine transformation layer for the BNN is the same as the general affine transformation. The only difference is that we use binarised activations, weights, and biases. The n-dimensional vector can be mapped to the m-dimensional vector using the affine transformation. The mathematical expression is:

$$Affine\_Transformation(x) = \sum_{i=1}^{n}(bin\_activation_i \times bin\_weight_i) + bin\_bias \tag{35}$$

1. Receive the binarised weights, biases and activations from the binarisation function.
2. Multiply the binarised activations and the weights and then add the bias for the dimension conversion of the vectors.

The affine transformation code is implemented based on the mathematical expression in Python. A linear function provided by Pytorch is used, and then the bias values are added to the value.

$$AT\_Layer = torch.functional.linear(input, weight) + bias \tag{36}$$

3.2.5 The ReLU layer

The Relu function is chosen for the activation function of the BNN to avoid gradient vanishing problems and increase the verification speed. The mathematical expression is:

$$Relu(x) = \begin{cases} x, & \text{if } x \geq 0. \\ 0, & \text{if } x < 0. \end{cases} \tag{37}$$

1. Receive the values from the affine transformation layer.

2. Put the input value into the ReLU activation function and convert it to 0 when it is negative and to the same input value when it is positive.

In Python, we can simply create the ReLU layer using the ReLU function provided by PyTorch.

$$ReLU\_Layer = torch.ReLU() \tag{38}$$

3.2.6 The Batch Normalisation layer

The batch normalisation for the BNN is the same as the general batch normalisation. The mathematical expression is:

$$\frac{x - E(x)}{\sqrt{V(x) + \epsilon}} * \gamma + \beta \tag{39}$$

1. Receive the values from the ReLU layer.
2. Subtract the average values $E(x)$ from the original values x.
3. Add the epsilon ϵ value to the denominator to avoid division by zero errors and to increase accuracy. Next, apply the square root to this value. epsilon ϵ is set to a default value of 0.00001.
4. The numerator is divided by the denominator and then multiplied gamma γ, adding beta β. Gamma γ defaults to 1, and beta β defaults to 0.

In Python, we can create a batch normalisation layer using the batchNorm1d function provided by PyTorch.

$$BN\_Layer = torch.BatchNorm1d() \tag{40}$$

3.2.7 Optimisation and Loss function

We choose Adam optimiser for finding the most appropriate weights and biases with a cross-entropy loss function. The default learning rate for Adam was set to 0.01.

- Optimiser: Adam
- Loss function: Cross entropy loss
- Learning rate: 0.01

We can set up an optimiser and loss function using the CrossEntropyLoss, Adam function provided by PyTorch in Python.

$$loss\_function = torch.CrossEntropyLoss() \tag{41}$$

$$optimiser = torch.optim.Adam(model.parameters(), learning\_rate) \tag{42}$$

3.2.8 Backward Propagation

In Python, backward propagation is performed based on the pre-defined loss function and optimiser using the backward function provided by PyTorch. The $output$ is the predicted values from the BNN, and the $target$ is the actual values from the MNIST. We obtain the loss based on the cross entropy loss function and then update the parameters through the backward and step functions.

$$loss = loss\_function(output, target) \tag{43}$$

$$loss.backward() \tag{44}$$

$$optimizer.step() \tag{45}$$

3.3 Testing

We choose the C language to write code to verify BNN forward propagation in ESBMC. BNN algorithms have emerged to employ machine learning in a restricted resource environment. C language is the most suitable language for embedded environments since C requires relatively smaller resources because it uses less memory and CPU than other languages. In addition, ESBMC supports various methods for C-language verification. Among them, we utilise assume and assert functions to ensure that BNNs satisfy the safety properties.

The MNIST data is used for the testing phase as well. We implemented the 3-layer BNN, 5-layer BNN and 7-layer BNN. The number of neurons in the input layer was fixed at 28*28 according to MNIST image data. The number of neurons in all hidden layers is set the same, and the number of neurons in the hidden layer is increased to 10, 50, 100, 150, and 200. For example, a BNN with three layers and 100 neurons has 784x100x100 neurons.

The BNN used in the test code has the same structure as the BNN trained in Python. In other words, both BNNs have the same number of layers and neurons with the same structure. The input layer and hidden layer consist of a binarisation layer, an affine transformation layer, a batch normalisation layer, and a ReLU layer, and the output layer only has the ReLU layer.

The test code is verified on ESBMC by checking if the BNN satisfies safety properties conditions. If the conditions are satisfied, the verification is successful and safety properties are satisfied. If these

conditions are not met, the verification is failed and the safety properties are dissatisfied.

Algorithm 2 illustrates the entire process of the testing period.

Algorithm 2 The algorithm of the testing phase

```
1:  procedure SET THE BNN MODEL
2:      for Every layer in the input layer and hidden layers do
3:          add binarise layer
4:          add affine transformation layer
5:          add ReLU activation layer
6:          add batch normalisation layer
7:      end for
8:      add ReLU activation layer for the output layer
9:  end procedure
10:
11: initialise the input data for ESBMC
12: assume the input data for ESBMC
13: set the BNN model
14: do forward propagation
15: find the index of maximum output data
16: asser that the output value with the maximum index is the biggest
```

A detailed description of the implementation of each layer of the test code is as follows.

3.3.1 Assume input data

The input data is the MNIST data, which is 28*28 size handwritten image data. Therefore, the input data size of the ESBMC is always 784. We the use binarised MNIST image data; therefore, the input data type can be declared as boolean.

We adopted the two functions from ESBMC:

- __ESBMC_init_object: ESBMC initialises memory objects
- __ESBMC_assume: If a condition is false, ESBMC aborts the execution.

We utilised the two function from ESBMC as follows:

- __ESBMC_init_object function: Declare that the input data value is a boolean data type.
- __ESBMC_assume: The maximum and minimum values of the input value data are set as the input values of the ESBMC. Since input data is binarised, the input value is 0 or 1. Based on these inputs, the conditions of the assert function are verified.

Algorithm 3 explains how to assert the input data using __ESBMC_init_object and __ESBMC_assume for ESBMC.

Algorithm 3 Assume input data

Input the MNIST image input data
Output esbmc data

```
ESBMC_init_object(esbmc_data)
for i = 1, 2, ... , 784 do
    if input_data[i] &&  1 then
        ESBMC_assume(esbmc_data[i] == 1)
    else
        ESBMC_assume(esbmc_data[i] == 0)
    end if
end for
```

3.3.2 The Binarisation Layers

The binarisation layer uses the deterministic binarisation function to binarise integer-valued or real-valued input. The deterministic binarisation function returns 1 if the input is bigger than 0 and returns 0 if the input is smaller than 0. The principal advantage of using binarised values is that they can be declared as a boolean type in C, which results in faster verification. Furthermore, we declare the binarisation function as a macro function to speed up computation and verification. The macro function is as follows:

$$\#define\, BIANRISE(X)\, (X >= \ ?\, 1 : 0) \tag{46}$$

3.3.3 The Affine Transformation layer

The affine transformation layer receives binary activations, weights, and biases as input values. The affine transformation layer multiplies activation by weights and then adds bias to convert the input data vector's dimension to the output vector's dimension. Since the affine transformation is performed only by addition and multiplication using the input values consisting of 0 and 1, the output values of the layer are in the form of an integer. Algorithm 4 describes the algorithm of the affine transformation layer for the input layer, and Algorithm 5 describes the algorithm of the affine transformation layer for the hidden layers. The two algorithms differ in the type of input data. In particular, the affine transformation layer for the hidden layers requires a procedure of binarising the input value.

The affine transform layer maps a vector of size left layer row size to right layer column size. Thus, it is necessary to multiply the input_data vector and the weights vector for the mapping. This study uses an AND operator instead of a multiplication operator. Statistically, bitwise operators are known as the fastest operators. Therefore, the verification time can be reduced by replacing the multiplication operator with the and operator. This substitution is only possible because the values of the input_data vectors and weights vectors are boolean data types consisting only of 0 and 1. Table 2 shows the result of the multiplication operation and the result of the AND($\&$) operation. As shown

Algorithm 4 The Affine Transformation layer for the input layer

Input esbmc data
Output affine layer output

for $i = 1, 2, ...$esbmc data row **do**
  for $j = 1, 2, ...$affine layer column **do**
    affine_layer_output[i][j] = bias[i][j]
    for $k = 1, 2, ...$esbmc data column **do**
      output[i][j] += (esbmc_data[i][k] & weights[k][j])
    end for
  end for

Algorithm 5 The Affine Transformation layer for the hidden layers

Input batch normalisation layer
Output affine layer output

for $i = 1, 2, ...$batch normalisation layer row **do**
  for $j = 1, 2, ...$affine layer column **do**
    affine_layer_output[i][j] = bias[i][j]
    for $k = 1, 2, ...$batch normalisation layer column **do**
      bin_batch_norm_layer_output[i][k] = BINARISE(batch_norm_layer_output[i][k])
      output[i][j] += (bin_batch_norm_layer_output[i][k] & weights[k][j])
    end for
  end for

in Table 2, since the results of the two operators are the same, the and operator can be used instead of the multiplication operator.

X	Y	Result(multiplication)	Result(AND)
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Table 2. The results of the multiplication and AND operator for boolean values

3.3.4 The ReLU layers for the hidden layers

The ReLU layers for hidden layers use the ReLU activation function. The ReLU function returns the input value if the input is greater than 0 and returns 0 if the input value is less than 0. In particular, the Relu function outputs without modifying the input value's data type. For example, if the input value is an integer, an integer value is the output, and if the input value is a real-valued number, a real-valued number is output. In the hidden layer, the ReLU layer is declared right after the Affine Transformation layer, which consistently outputs integer values. Therefore, the ReLU layers for hidden layers always receive and output integer values. The ReLU function is also declared a macro function to speed up computation and validation. The macro function is as follows:

$$\#define\ ReLU(X)\ MAX(0, X) \tag{47}$$

Algorithm 6 describes the algorithm of the ReLU layers for hidden layers.

Algorithm 6 The ReLU layers for the hidden layers

Input affine layer output
Output relu layer output

1: **for** $i = 1, 2, ...$neurons **do**
2: relu_layer_output[i] = ReLU(batch_norm_layer_output[i])
3: **end for**=0

The output value of the affine transform layer is received as an input value, which is converted through the ReLU macro function. The resulting value is stored in relu_layer_output.

3.3.5 The Batch Normalisation layers

In order to increase accuracy, we implement the batch normalisation layer in the same way as the training phase. The general mathematical expression of batch normalisation is:

$$\frac{x - E(x)}{\sqrt{V(x) + \epsilon}} * \gamma + \beta \tag{48}$$

Here, we set gamma to 1, beta to 0, and epsilon to 0.00001. The batch normalisation formula can be simplified with these default values:

$$\frac{x - E(x)}{\sqrt{V(x) + 0.00001}} \tag{49}$$

We use the lookup table method to speed up calculation and verification. Since square root and division are operations with high computational complexity in the c language,

$$modified\_variance(x) = \frac{1}{\sqrt{V(x) + 0.00001}} \tag{50}$$

is calculated in advance and declared constant. The E(x) values obtained in the training phase are also declared constants in advance. Algorithm 7 describes the algorithm of the batch normalisation layer.

Algorithm 7 The Batch Normalisation layers

Input relu layer output
Output batch normalisation layer output

1: **for** $i = 1, 2, ...$neurons **do**
2: batch_norm_layer_output[i] = (relu_layer_output[i] - mean[i]) * modified_variance[i]
3: **end for**

The batch normalisation layer receives the output from the ReLU layer. The modified_variance is the lookup table for $\frac{1}{\sqrt{V(x)+0.00001}}$. We can normalise the batch data by multiplying the pre-calculated modified variance by a value obtained by subtracting the mean value from the ReLU layer output value.

3.3.6 The ReLU layers for the output layer

The ReLU algorithm and function of the output layer are the almost same as the ReLU activation function and algorithm used in the hidden layer. However, since the last layer before the output layer is the batch normalisation layer, there is a difference in that the output value is real-valued. Since MNIST data is an image with 0-9 written in handwriting, the BNN can predict the number of the input image data using the output values. In order to obtain the predicted value, we need to find an index having the largest value in the output value of the output layer. Algorithm 8 describes the process of the output ReLU layer and obtaining the predicted value.

Algorithm 8 The ReLU layers for the output layer

Input batch normalisation layer output
Output relu layer output

1: **for** $i = 1, 2, ...$neurons **do**
2: relu_layer_output[i] = ReLU(batch_norm_layer_output[i])
3: **end for**
4: find_max_index(relu_layer_output)

The output value of the batch normalisation layer is used as the input value of the ReLU output layer. The output value of the ReLU output layer is obtained by using the ReLU function in the same way as the hidden layer. Next, as the MNIST number consists of 0 to 9, 10 repetitions are required to predict the number. The index with the largest value among the output values is the predicted number through the BNN.

3.3.7 Assertion

In BNN, we can find the index with the largest value in the output value, and this index is the predicted number. Therefore, we can use this to determine whether this BNN satisfies the safety properties. The predicted number with the largest output value must have the same value as the target number to satisfy the safety properties. For example, the input data value is an image of handwriting 6. The numbers 0 and 6 are likely to be incorrectly predicted because they look similar. If the predicted output value of index 6 is always greater than other output values, the safety properties are satisfied. However, if another index, such as 0, has the maximum value of the output layer, the output value of this BNN does not satisfy the safety properties. Algorithm 9 describes how to assert the safety properties.

We adopted the assert functions from ESBMC:

- __ESBMC_assert: If a condition is false, ESBMC ignores the execution.

We utilised the assert function from ESBMC as follows:

- __ESBMC_assert: check if the index with the highest output value from the output layer matches the goal value, the actual number of data in the MNIST test.

Algorithm 9 Assertion

Input BNN output

1: **for** $i = 1, 2, ... 9$ **do**
2: **if** i != predicted_number **then**
3: ESBMC_assert(output[predicted_number] $>=$ output[other$_{n}umber$]) **end if**
4:

The $output$ values with indexes from 0 to 9 are compared to ensure that the predicted values are always greater than or equal to the output values of other indexes. The same index as the $predicted\_number$ is excluded because there is no need to compare. Since the final output value is derived through the ReLU function, there is a probability that all output values become 0. Therefore, we define that safety properties are satisfied even when the output value with the index of the predicted value is the same as the output value of other indexes. If the output value with the predicted index is the maximum, safety properties are met, which results in verification success. Otherwise, the verification is a failure.

3.4 Verification on ESBMC

3.4.1 Testing on ESBMC

In order to verify the c test code on ESBMC, we can use the esbmc command in the environment where ESMBC is installed. We can also select the solvers among Boolector, Z3, MathSAT, CVC4, Bitwuzla, and Yices. The default solver is Boolector. The ESBMC checks whether the verification is successful and provides information about the verification. The information provided by the ESBMC is as follows.

- Verification Result: success, failure
- Time Information: symex completed time, Slicing time, Encoding to solver time, Runtime decision procedure, BMC program time

- Assignment Information: Generated VCC(s), the number of assignments, the number of assignments after simplification
- Counterexample: provided only if the verification was failed
- Violated Properties: provided only if the verification was failed

We focus on successful verification and measure and compare encoding to solver time, runtime decision procedure, BMC program time, and the number of assignments among the metrics provided by ESBMC.

We also compare maximum memory usage for each verification by adopting the Linux command sar. We check the amount of memory in use every 1 seconds through the Linux command sar and obtain the maximum value of memory usage. The command is:

$$sar\ -r\ 1 \tag{51}$$

3.4.2 Solver Selection

There are various types of solvers that ESBMC supports: Boolector, Z3, MathSAT, CVC4, Bitwuzla, and Yices. Solvers should meet two criteria. First, the solver should support floating point verification. CVC4 is excluded because it does not support the floating point operation. Second, the solver should not have a significantly slower verification time than other solvers. Since we find that the Boolector, Yices, and Bitwuzla solvers complete the verification much faster than MathSat and Z3 solvers from several experiments, the Boolector, Yices, and Bitwuzla solvers are selected. For example, when verifying a 748x10x10 size BNN, the Boolector, Yices, and Bitwuzla solver completed the verification in approximately 12 seconds. However, the MathSat and Z3 solver took more than 25 seconds, which was more than twice the verification time of Boolector, Yes, and Bitwuzla solver.

3.5 Benchmarks

We take the average value of the benchmark obtained by verifying BNNs with the same number of layers and the same neuron in the test environment three times in total on ESBMC. Since the values of encoding to solver time, runtime decision procedure, BMC program time, and maximum memory usage change depending on the type of solver, we also conduct the experiments for each Boolector, Yices, and Bitwuzla solver. The number of assignments value is constant regardless of the type of solver.

3.6 Summary

To conclude, to test BNNs on ESBMC, we implement a BNN forward propagation algorithm using assume and assert statements to check safety properties are satisfied in C. ESBMC is installed in the testing environment, and benchmarks are measured while testing in this environment.

4 Results and discussion

4.1 Introduction

We discuss the results of the verification test of the pre-trained BNN. The layer size of the BNN is raised to 3, 5, and 7. In terms of the number of neurons, the first layer's input size is fixed to the image size 28*28 of the MNIST data. The test is conducted while increasing the number of neurons in the hidden layer from 10 to 200. The number of output layer neurons equals the number of neurons in the previously hidden layer.

We measure BMC program time which is the total verification time, the number of assignments, encoding to solver time, and the runtime decision procedure among the metrics provided by ESBMC. We also check the maximum memory usage using the Linux command sar. Furthermore, we compare the metric values that vary with solver by changing the type of solver to Boolector, Bitwuzla, and Yices. The test environment is as follows:

- Environment: Virtual Machine
- CPU: 8 cores, 16 threads
- Memory: 12GB
- OS: Linux Mint 20.2 Uma

4.2 Experiments

The structure of each BNN for testing is a fully-connected neural network. The input layer and the hidden layer are composed of affine transformation, batch normalisation, and ReLU, and the output layer is composed of only ReLU. The default epoch is set to 100, and the batch size for training is set to 64. We choose Adam optimiser with a batch normalisation method for the training period for each batch set. We verify the safety properties of the pre-trained BNNs while testing the forward propagation through ESBMC.

The data used in this experiment is an image of MNIST data handwritten numeric images from 0 to 9. By selecting one of these images, we can set the safety properties to check if the pre-trained BNNs predict the image correctly, and then the ESBMC verifies it. In addition, since this study aims to verify BNN within a reasonable time, BNN should be verified within an hour (60min) on ESBMC. We measure each metric three times and then obtain the average value. The metrics offered by ESBMC are as follows:

- BMC program time: Total time spent validating BNNs on ESBMC
- Encoding to solver time: Time spent encoding to send to Solver
- Runtime decision procedure: Time spent verifying SMT formulae by a solver
- assignments: Total number of assignments required for verification

In addition, we also measure the maximum memory usage for each experiment from Linux.

4.3 Case study

We perform the experiments while increasing the number of layers to 3, 5, and 7. The number of neurons in the input layer is 28*28. The number of neurons in the hidden layer increases to 10, 50, 100, 150, and 200. Furthermore, the metrics for the Boolector, Bitwuzula, and Yices solver are respectively measured. We also measure memory usage while performing verification on the ESBMC. Regardless of the type of solver, memory usage shows a remarkably similar tendency. Memory usage grows gradually before solving with a solver. After the solver starts to solve formulae, memory usage rapidly increases. There is a peak just before the solver completes the verification.

4.4 3-layer BNNs

4.4.1 The structure of the 3-layer BNNs

The 3-layer BNN structure used for the test is shown in Figure 11. It consists of one input layer, one hidden layer, one output layer, and a total of three layers. The input layer and the hidden layer are composed of binarisation, affine transformation, ReLU activation function, batch normalisation, and the output layer is composed of one ReLU activation function.

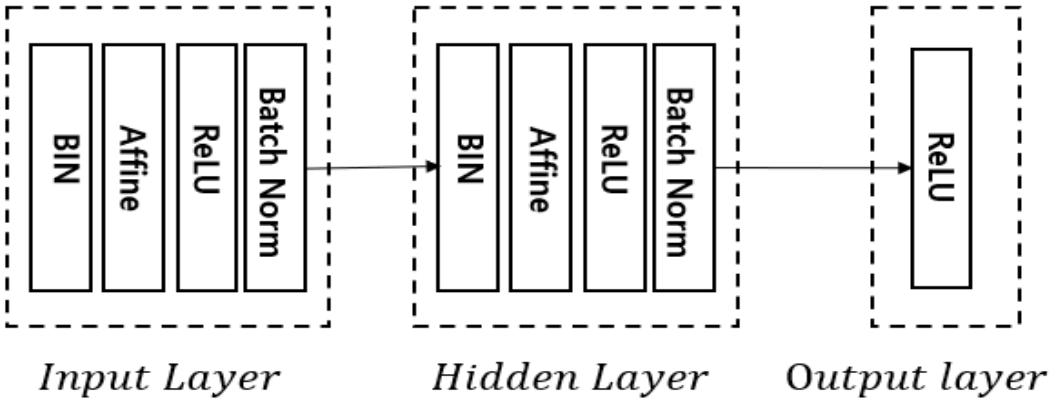

Fig. 11. The structure of the 3-layer BNN

4.4.2 The number of assignments

Fig 12 shows the number of assignments. The number of assignments linearly increases as the number of neurons in the hidden layer increases. In general, an increase in the number of assignments corresponds to an increase in the number of neurons in the hidden layer. While the number of neurons expanded fivefold from 10 to 50, the number of assignments increased around 5.05 times. While the number of neurons doubled from 50 to 100, the number of assignments grew by around 2.14 times. The number of assignments grew nearly 2.3 times as the number of neurons increased from 100 to 200. According to the experiment, the assignments tend to increase proportionately to the number of neurons in the hidden layer.

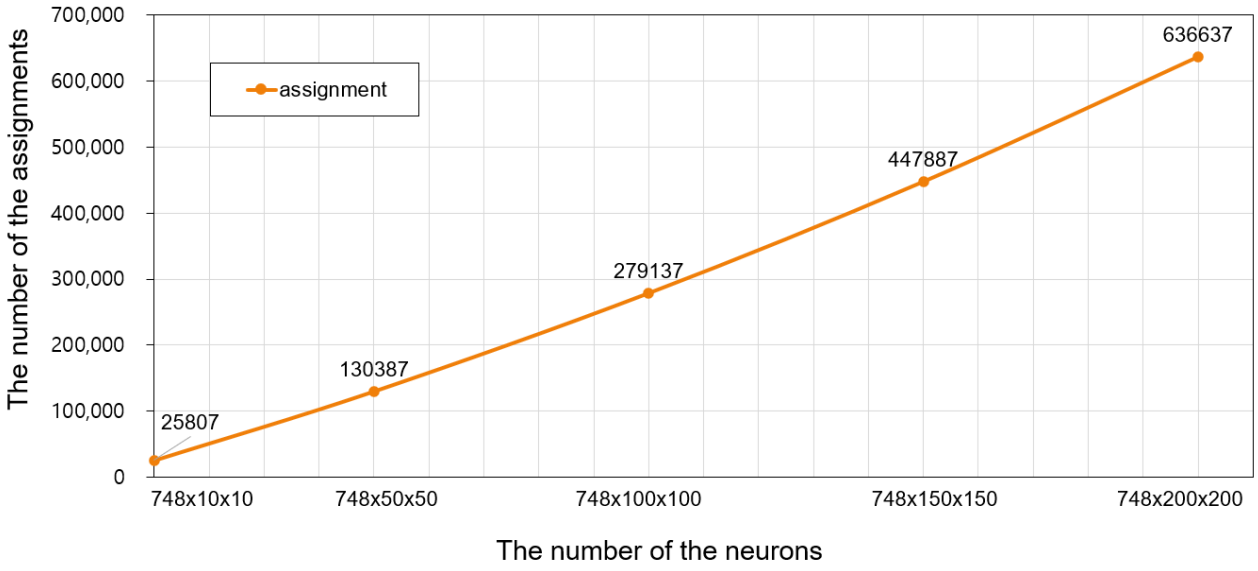

Fig. 12. The number of assignments

4.4.3 The maximum memory usage

Table 3 and Fig 13 illustrates the maximum memory usage. When verifying the 3-layer BNNs in all three solvers, the maximum memory usage tends to be similar. When the number of neurons increased ten times from 10 to 100, the average memory usage increased nearly 7.2 times from 3.39% to 24.36%. In addition, when the number of neurons doubled from 100 to 200, the average memory usage also doubled from 24.36% to 52.45%. In conclusion, as the number of neurons increases, memory usage increases more quickly, but the speed is not remarkably fast. As a result, in 3-layer BNN, the total usage does not exceed 55%, regardless of the solver.

The number of neurons	Boolector (%)	Bitwuzla(%)	Yices(%)	Average(%)
748x10x10	3.37	3.34	3.46	3.39
748x50x50	12.89	12.25	12.26	12.47
748x100x100	25.15	24.52	23.42	24.36
748x150x150	38.17	37.98	36.9	37.68
748x200x200	52.64	53.88	50.83	52.45

Table 3. The maximum memory usage

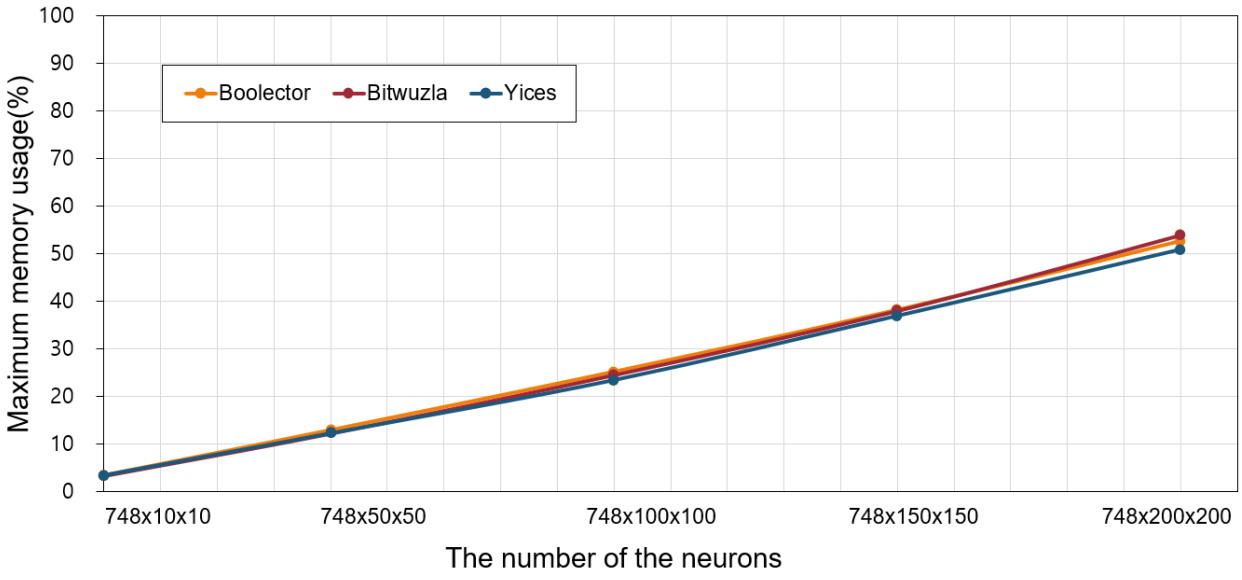

Fig. 13. The maximum memory usage

4.4.4 The verification time on Boolector

Table 4 and figure 14 show the BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Boolector solver.

Overall, as the number of neurons increases, the verification time increases exponentially. While the number of neurons increased ten times from 10 to 100, the verification time increased by 44

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10	0.191	0.030	0.011
748x50x50	2.232	1.275	0.074
748x100x100	8.752	6.436	0.173
748x150x150	21.241	17.033	0.317
748x200x200	32.507	26.252	0.484

Table 4. The verification time on Boolector

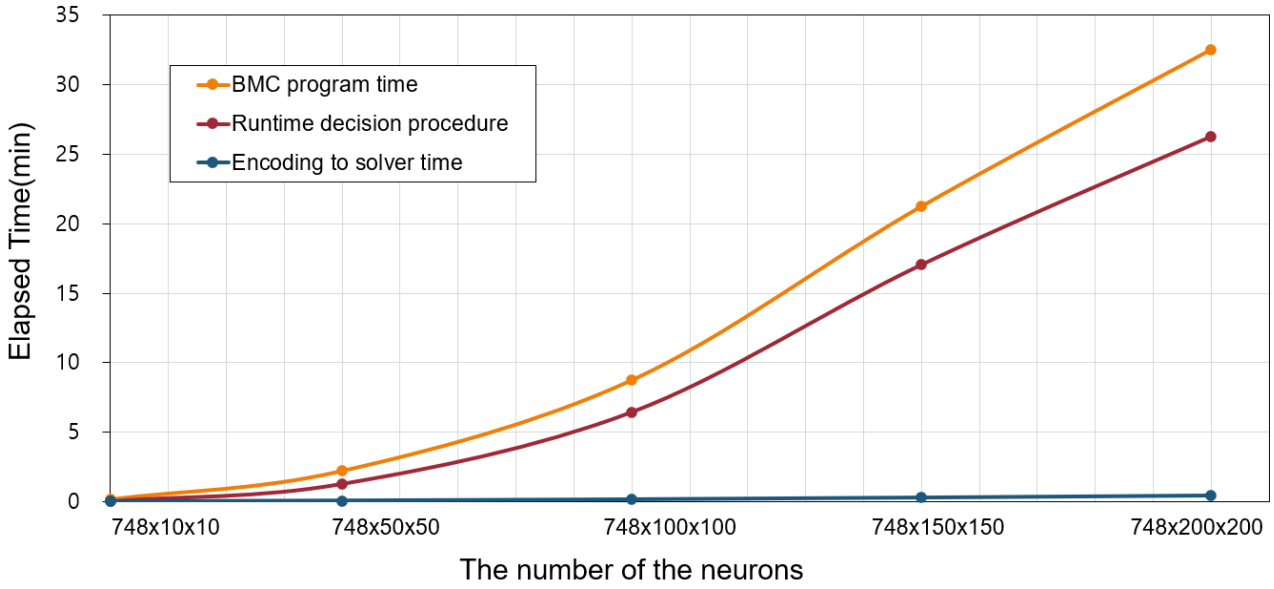

Fig. 14. The verification time on Boolector

from 0.191 minutes to 8.752 minutes. The verification time increased approximately fourfold from roughly 8.752 minutes to 32.507 minutes, while the number of neurons doubled from 100 to 200. The main bottleneck of this surge is the runtime decision procedure time. The runtime decision procedure time accounts for approximately 80% of the verification time when the number of neurons in the hidden layers exceeds 100. Since all the encoding to solver time of the Boolector solver measured is less than 1 minute, this metric does not significantly affect the total verification time.

4.4.5 The verification time on Bitwuzla

Table 5 and Figure 15 show the BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Bitwuzla solver. The Bitwuzla solver has metrics that are closely comparable to the Boolector. Furthermore, the primary bottleneck of the Bitwuzla solver is also the runtime decision procedure, and the encoding to solver time is minimal.

The Bitwuzla solver also shows that the verification time increases exponentially as the number of neurons increases. While the number of neurons increased tenfold from 10 to 100, the verification time increased 44 times from 0.192 to 8.628 minutes. Furthermore, while the number of

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10	0.192	0.031	0.011
748x50x50	2.237	1.281	0.077
748x100x100	8.628	6.315	0.174
748x150x150	21.844	17.566	0.340
748x200x200	32.429	26.163	0.450

Table 5. The verification time on Bitwuzla

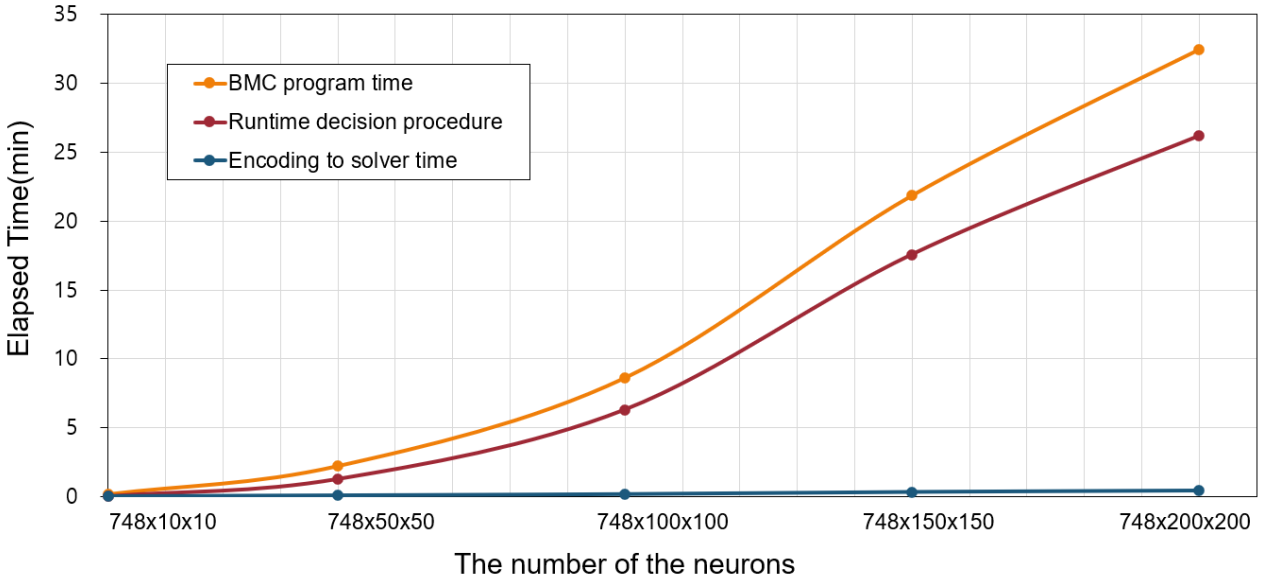

Fig. 15. The verification time on Bitwuzla

neurons doubled from 100 to 200, the verification time nearly quadrupled from 8.628 minutes to 32.429 minutes. The runtime decision procedure time accounts for about 80% of the verification time when the number of neurons is more than 100. Because all the encoding to Solver time of Bitwuzla measured in the experiment is less than 1 minute, this metric is trivial.

4.4.6 The verification time on Yices

Table 6 and Figure 16 show the BMC program time, runtime decision procedure, and encoding to solver time, measured by ESBMC while increasing the number of neurons on the Yices solver. The Yices solver produces metrics that differ from those produced by the Bitwuzla solver and the Boolector. In particular, if the number of neurons in the hidden layer is more than 200, the verification time takes only half. In addition, unlike Boolector and Bitwuzla solver, encoding to solver time is the main bottleneck point, and the runtime decision procedure does not take as long as encoding to solver time.

Even though the Yices solver is faster than the Bitwuzla and the Boolector solvers, the verification time also grows exponentially with the number of neurons. While the number of neurons increased tenfold from 10 to 100, the verification time increased by 20 times from 0.168 to 3.466

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10	0.168	0.005	0.014
748x50x50	1.132	0.021	0.227
748x100x100	3.466	0.039	1.286
748x150x150	9.008	0.121	4.988
748x200x200	15.543	0.098	9.460

Table 6. The verification time on Yices

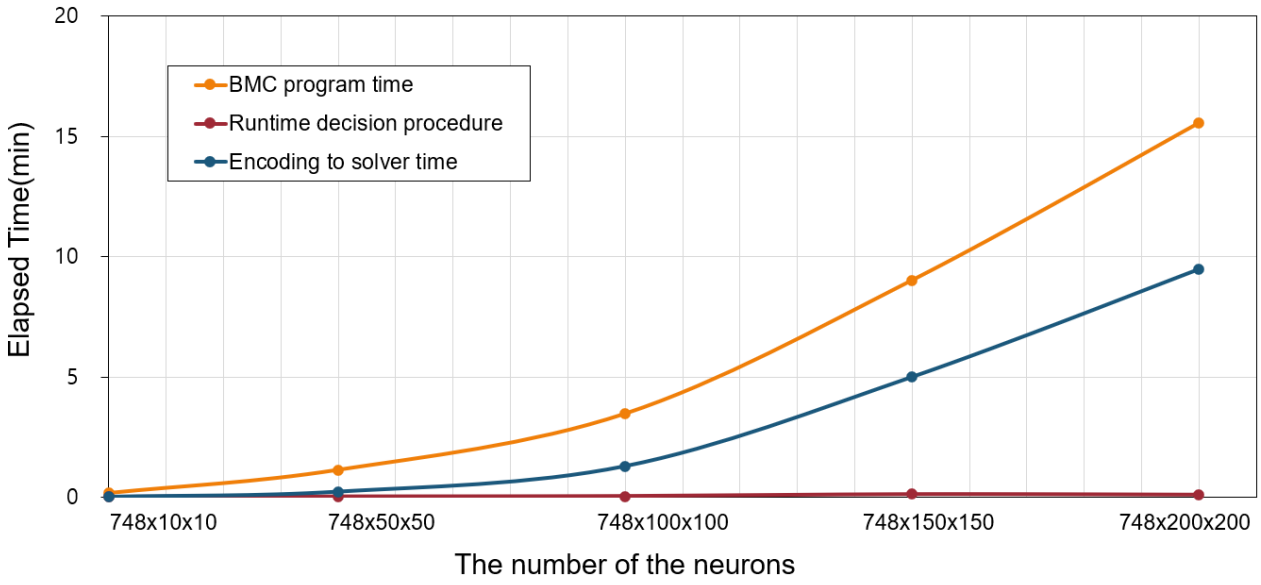

Fig. 16. The verification time on Yices

minutes. Furthermore, as the number of neurons doubled from 100 to 200, the verification time rose approximately 4.5 times, from 3.466 minutes to 15.543 minutes. The Yices solver is the fastest and can handle scalability better than other solvers in the 3-layer BNN experiments. Unlike other solvers, the main bottleneck of the Yices solver is encoding to solver time. If the number of neurons exceeds 100, it accounts for more than half of Yices' total verification time. On the other hand, the runtime decision procedure time accounts for less than 1% of the total verification time.

4.4.7 Comparison between the three solvers

Fig 17 shows the BMC program time measured in the three solvers. The verification times of the Boolector and Bitwuzla solver follow a similar pattern. When the number of neurons is less than 50, the Yices solver also tends to be similar. However, as the number of neurons increases, the Yices solver can complete the verification roughly twice as fast as other solvers. According to the experiments for the 3-layer BNNs, the Yices solver performs the best. If the number of neurons is small, we can use any solvers among them, but if the number of neurons is large, using the Yices solver is appropriate for the 3-layer BNNs.

Fig 18 describes the runtime decision procedure time measured in the three solvers. Boolector

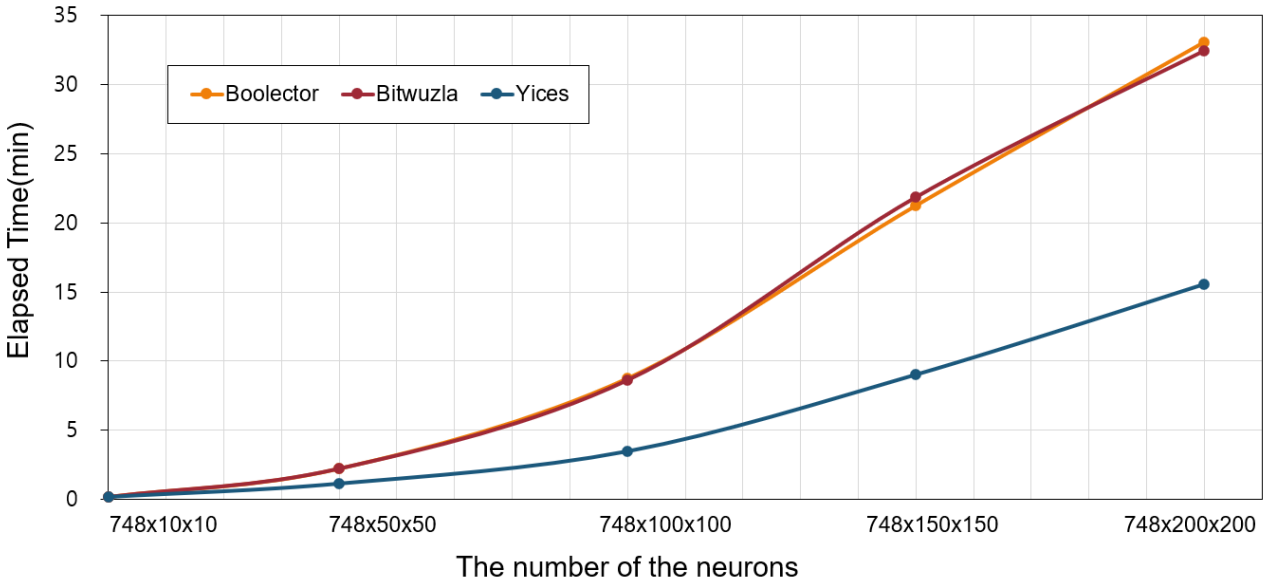

Fig. 17. The BMC program time

and Bitwuzla also exhibit similar patterns during runtime decision procedure time. When verifying BNNs based on Boolector and Bitwuzla, this metric accounts for the most time in total verification time, significantly when the number of neurons exceeds 100, accounting for more than 80% of total verification time. In the Yices solver, however, the runtime decision procedure time accounts for only 1% of the total verification time.

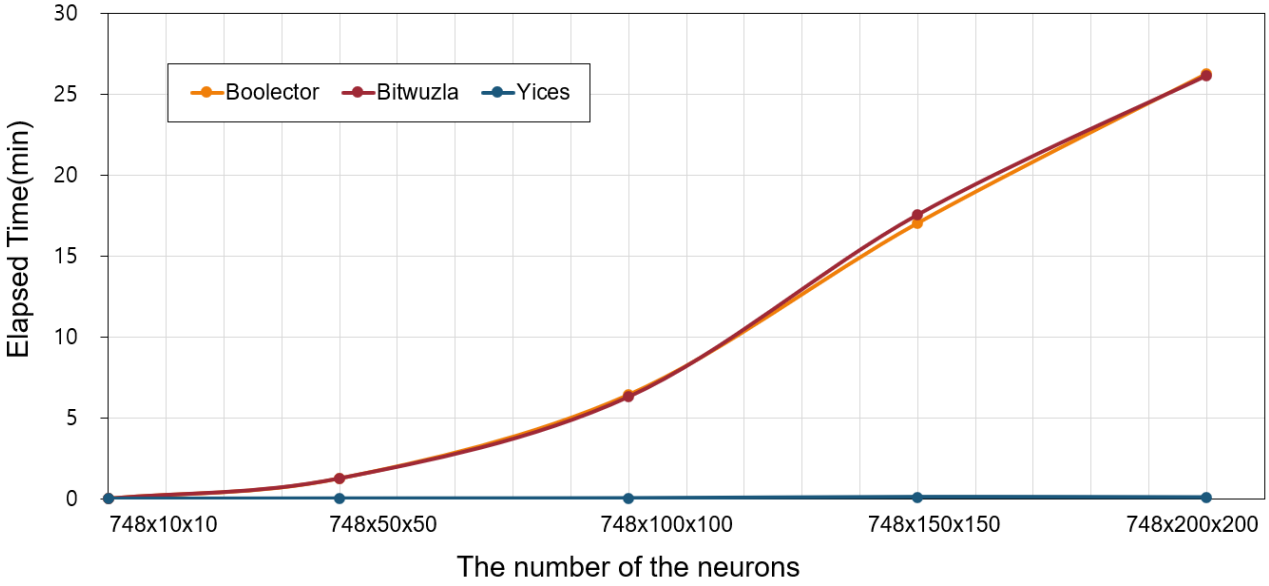

Fig. 18. The Runtime decision procedure time

Fig 19 illustrates the encoding to solver time measured in the three solvers. The encoding to solver time metric has a minor impact on the Boolector and Bitwuzla verification times. The encoding to solver time of the Boolector and Bitwuzla is less than one minute. However, it is a highly influential factor in the verification time of the Yices solver. The encoding to Solver time of Yices rapidly increases as the number of neurons increases.

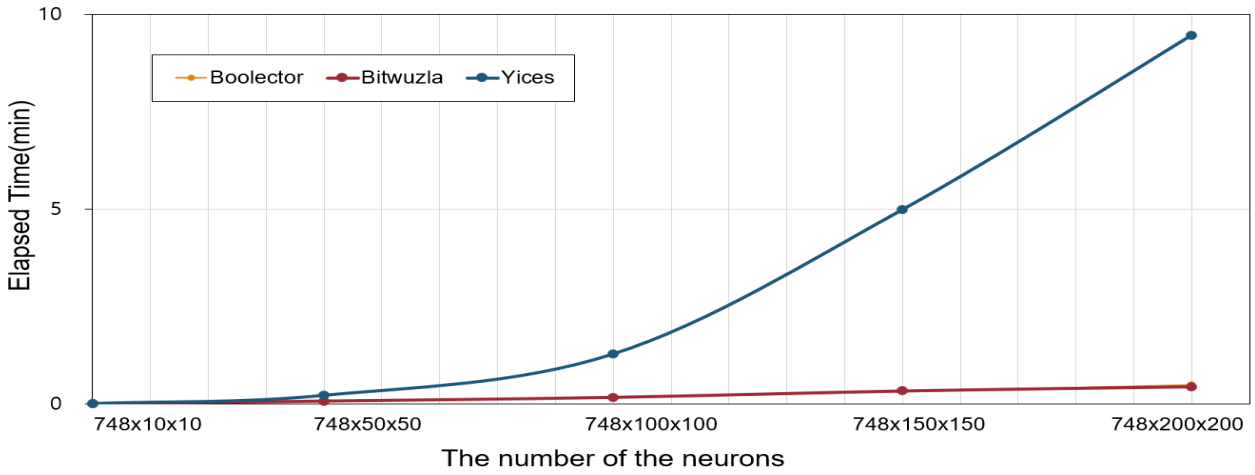

Fig. 19. The Encoding to solver time

4.5 5-layer BNNs

4.5.1 The structure of the 5-layer BNNs

The 5-layer BNN structure used for the test is shown in Figure 20. It consists of one input layer, three hidden layers, one output layer, and a total of five layers. The input layer and the hidden layer are composed of binarisation, affine transformation, ReLU activation function, batch normalisation, and the output layer is composed of one ReLU activation function.

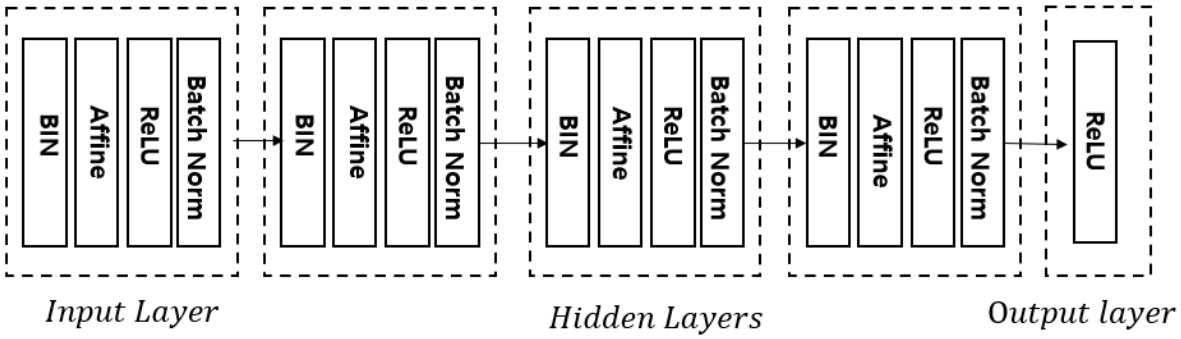

Fig. 20. The structure of the 5-layer BNN

4.5.2 The number of assignments

Fig 21 shows the number of assignments. Like the 3-layer BNN, the number of assignments increases considerably fast with the number of neurons in the hidden layer but still shows a linear pattern. However, the number of assignments grows substantially faster than the 3-layer BNN. The number of assignments roughly rose six times when the number of hidden layer neurons went from 10 to 50. The number of assignments increased 2.5 times as the number of neurons in the

hidden layer doubled from 50 to 100. When the number of neurons in the hidden layer doubled from 100 to 200, the number of assignments nearly tripled. The findings of this experiment show that as the number of neurons in the hidden layer of the 5-layer BNN increases, the gradient of the assignment becomes steeper than that of the 3-layer BNN, but it still appears a linear increase.

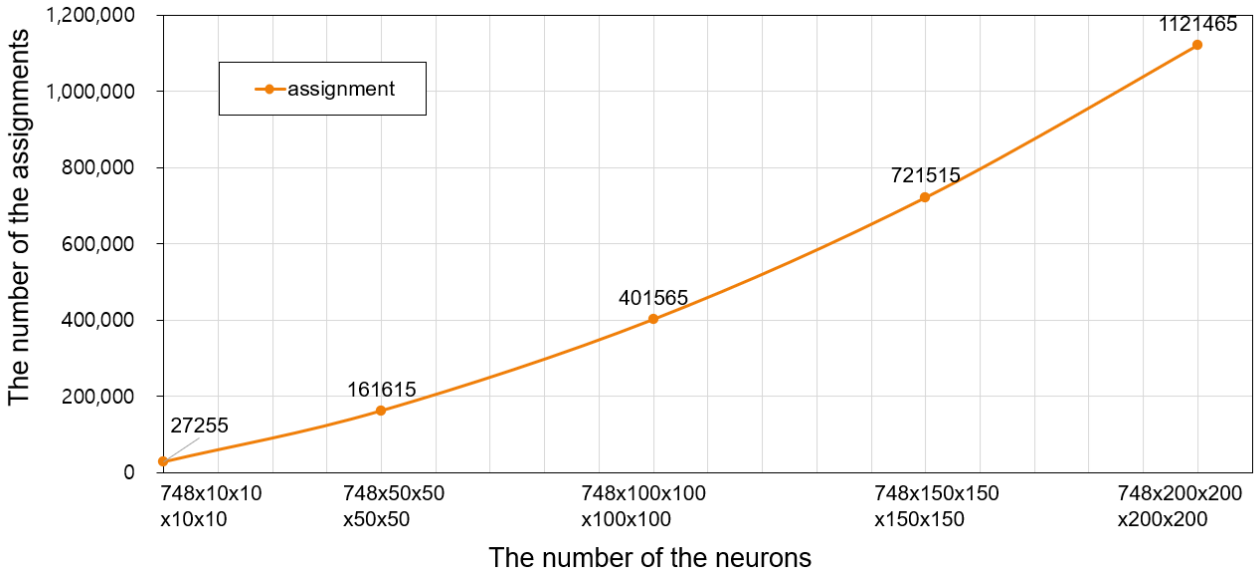

Fig. 21. The number of assignments

4.5.3 The maximum memory usage

Table 7 and Fig 22 illustrates the maximum memory usage. There is no significant difference in memory used to verify 5-layer BNNs for all solvers. Although the Yices solver tends to use a little more memory than other solvers, the difference is minor. When the number of neurons increased ten times from 10 to 100, the average memory usage also increased approximately ten times. When the number of neurons doubled from 100 to 200, the average memory usage grew by 2.5 times. In 5-layer BNNs, memory usage increases faster as the number of neurons increases, but the gradient of the increase is not steep. Most notably, as the number of neurons becomes 200, memory usage in all solvers is higher than 90%. With this high memory usage, if the number of neurons is increased by more than 200, significant degradation in performance or ESBMC verification failures due to memory shortages can be expected.

The number of neurons	Boolector (%)	Bitwuzla(%)	Yices(%)	Average(%)
748x10x10x10x10	3.42	3.47	4.14	3.68
748x50x50x50x50	14.98	15.3	17.91	16.06
748x100x100x100x100	34.6	35.58	40.95	37.04
748x150x150x150x150	60.24	61.41	69.73	63.79
748x200x200x200x200	92.97	93.72	93.29	93.33

Table 7. The maximum memory usage

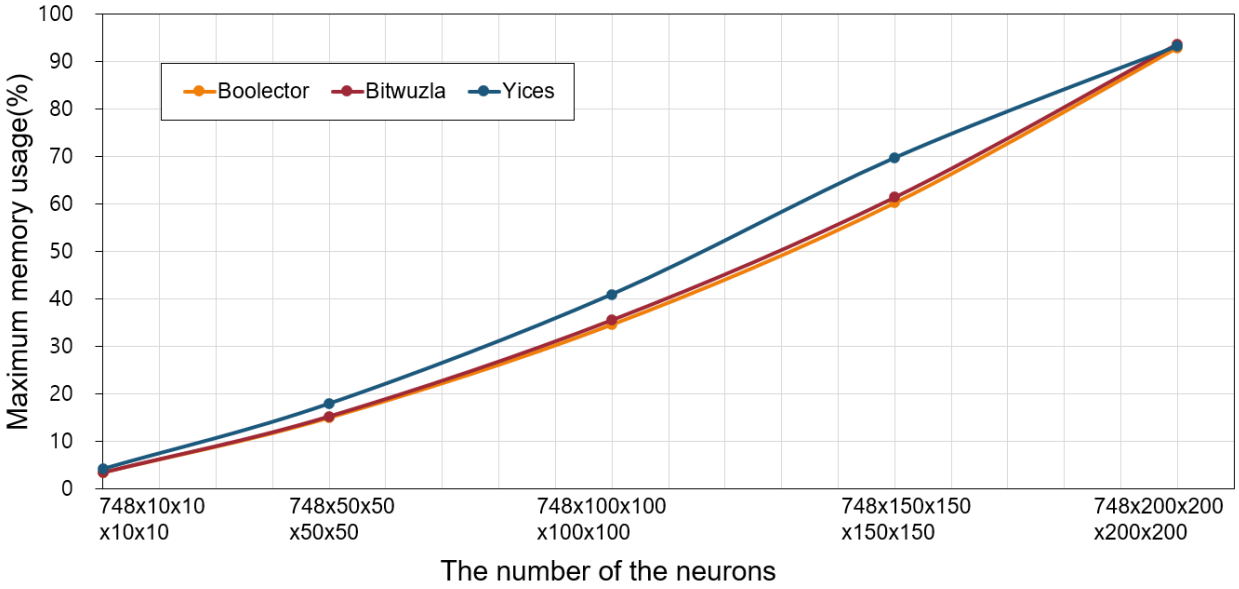

Fig. 22. The maximum memory usage

4.5.4 The verification time on Boolector

Table 8 and Figure 23 show the BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Boolector solver.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10	0.218	0.032	0.014
748x50x50x50x50	2.558	1.224	0.098
748x100x100x100x100	10.575	6.410	0.292
748x150x150x150x150	25.610	16.984	0.591
748x200x200x200x200	52.751	36.720	1.221

Table 8. The verification time on Boolector

When using the Boolector solver to verify the 5-layer BNN, the verification time grows exponentially with the number of neurons. When the number of neurons increased ten times from 10 to 100, the verification time increased approximately 49 times from about 0.218 minutes to roughly 10.575 minutes. Additionally, the verification time rose nearly five times, from 10.575 minutes to 52.751 minutes, when the number of neurons doubled from 100 to 200. Like the 3-layer BNN, the 5-layer BNN also rapidly grow in verification time as the number of neurons grows, but its growth speed is faster than 3-layer BNNs. If the number of neurons is more than 200, the runtime decision procedure time accounts for about 70% of the verification time, so it is the most critical factor in determining the verification time. Since all encoding to solver time in the experiment accounts for less than 1%, this metric does not significantly affect the total verification time.

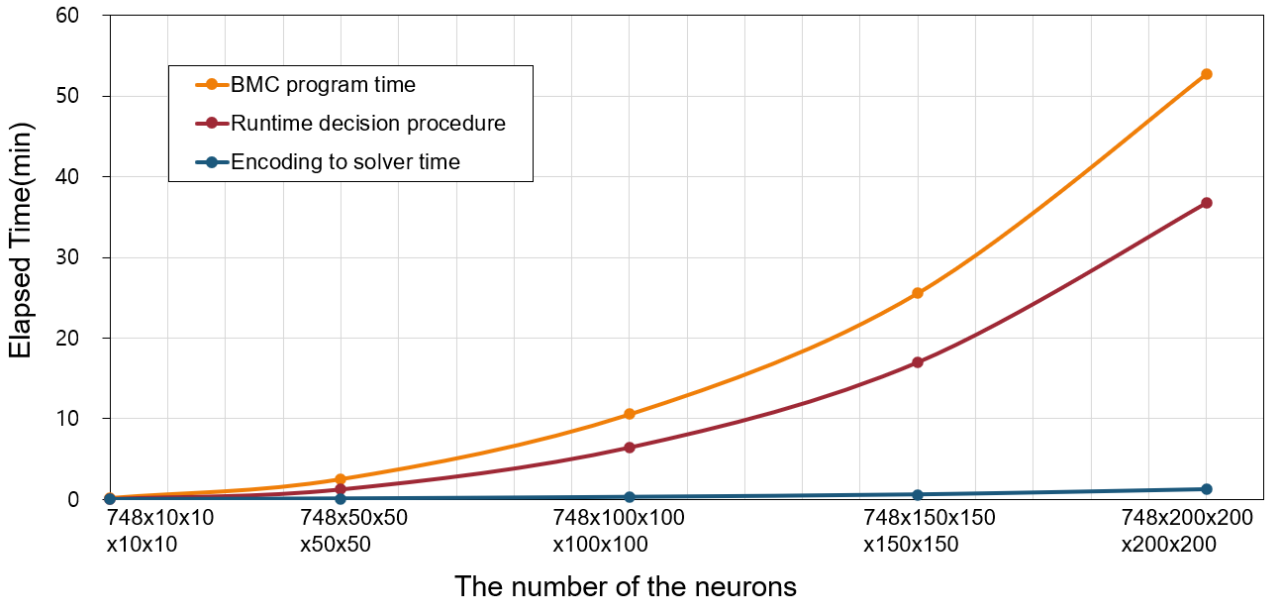

Fig. 23. The verification time on Boolector

4.5.5 The verification time on Bitwuzla

Table 9 and Figure 24 show the BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Bitwuzla solver.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10	0.217	0.034	0.014
748x50x50x50x50	2.714	1.314	0.111
748x100x100x100x100	10.671	6.563	0.297
748x150x150x150x150	25.123	16.626	0.572
748x200x200x200x200	56.899	41.000	1.219

Table 9. The verification time on Bitwuzla

The Bitwuzla solver also shows the growth of the verification time in exponential form as the total number of neurons increases, and the verification time is similar to the Boolector but takes a little longer. When the number of neurons increased ten times from 10 to 100, the verification time rose roughly 49 times from about 0.217 minutes to about 10.671 minutes. Additionally, the verification time increased by nearly 5.3 times, from about 10.671 minutes to about 56.899 minutes, when the number of neurons doubled from 100 to 200. The factor that accounts for most of the verification time is the runtime decision procedure time, just like the Boolector. If the number of neurons is more than 200, the runtime decision procedure time accounts for more than 72% of the total verification time. Because all encoding to solver time in the experiment accounts for less than 1% of the total verification time, this figure has little effect on the overall verification time.

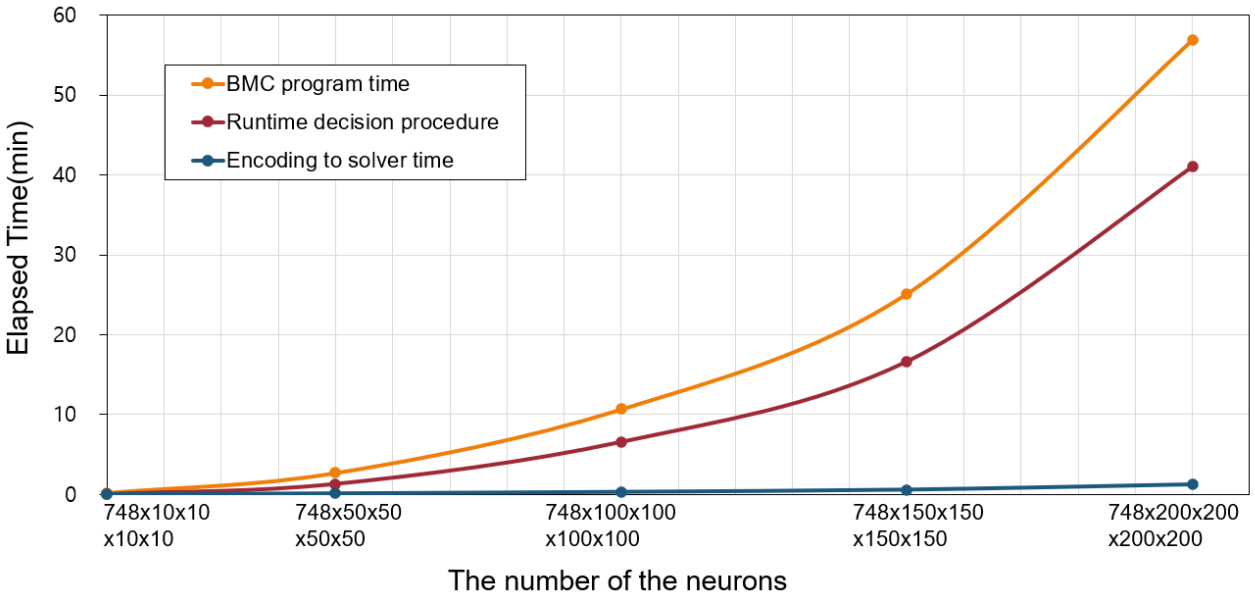

Fig. 24. The verification time on Bitwuzla

4.5.6 The verification time on Yices

The BMC program time, runtime decision procedure, and encoding to solver time measured by ESBMC while increasing the size of BNN neurons in the Yices solver are shown in Table 10 and Figure 25, respectively. Compared to the Bitwuzla and Boolector solvers, the Yices solver is the slowest, unlike when verifying 3-layer BNNs. Furthermore, when the number of neurons in the hidden layer is 200 or more, the verification time was 88.840 minutes, which took much longer than an hour. This experiment implies that the Yices solver cannot verify the 5-layer BNN with 200 neurons in a reasonable time, which is an hour. encoding to solver time is a significant bottleneck, unlike Boolector or Bitwuzla solver, and the runtime decision procedure is significantly faster than other solvers.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10	0.204	0.012	0.020
748x50x50x50x50	2.350	0.016	1.074
748x100x100x100x100	11.995	0.112	8.089
748x150x150x150x150	35.357	0.284	27.002
748x200x200x200x200	88.840	0.454	72.380

Table 10. The verification time on Yices

As the number of neurons increases, the verification time of the Yices solver grows the most rapidly compared to other solvers. The verification time increased nearly 59 times, from 0.204 minutes to 11.995 minutes, when the number of neurons went from 10 to 100. Additionally, the number of neurons doubled from 100 to 200, but the verification time increased by nearly 7.4 times, from 11.995 to 88.84 minutes. From this experiment, we can conclude that the Yices solver shows the slowest verification time if the number of layers increases compared to other solvers. Unlike other

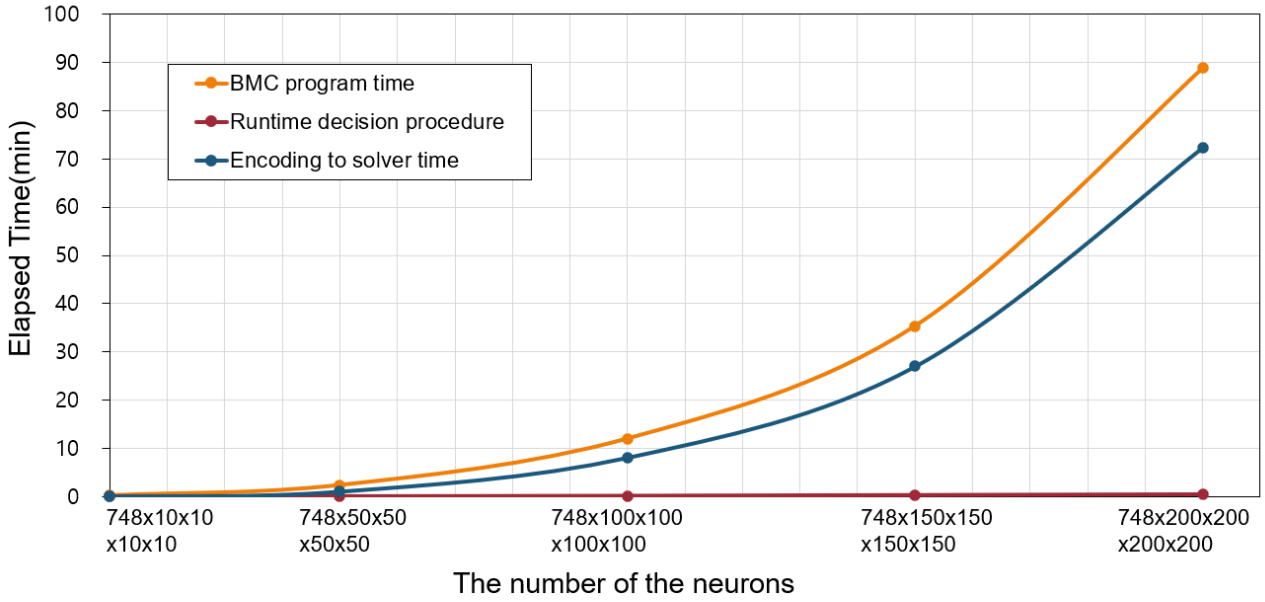

Fig. 25. The verification time on Yices

solvers, the main bottleneck in Yices solver is encoding to solver time, and if the number of neurons is more than 100, it accounts for more than 67% of Yices's total verification time. In contrast, the runtime decision procedure time is less than 1% of the total verification time.

4.5.7 Comparison between the three solvers

Fig 26 shows the BMC program time measured in the three solvers. Although the Boolector is slightly faster on the 5-layer BNN, the verification time between the Boolector and Bitwuzla is not different by more than 5 minutes, showing almost the same pattern. The Yices solver show similar patterns when the number of neurons is less than 100. However, if the number of neurons exceeds 100, the Yices solver's verification time increases significantly compared to other solvers. When there were 150 neurons in the hidden layer, the Boolector and Bitwuzla took approximately 25 minutes, but the Yices took roughly 35 minutes. The Boolector and Bitwuzla took about 53 and 57 minutes, respectively, when there were 200 neurons in the hidden layer, but Yices took nearly 89 minutes. Considering their verification speed and ability to handle scalability for the 5-layer BNN with more than 100 neurons, we can conclude that it is more reasonable to use the Boolectors or the Bitwuzla than the Yices for the 5-layer BNNs.

Fig 27 describes the runtime decision procedure time measured in the three solvers. The runtime decision procedure time is an element that accounts for the highest percentage of the total verification time of the Boolector and Bitwuzla. In particular, if there are more than 100 neurons, it accounts for more than 60%, and if there are more than 200 neurons, it accounts for more than 70%, and the proportion of this metric gradually increases. On the other hand, the runtime decision procedure time in the Yices solver does not account for 1% of the total verification time.

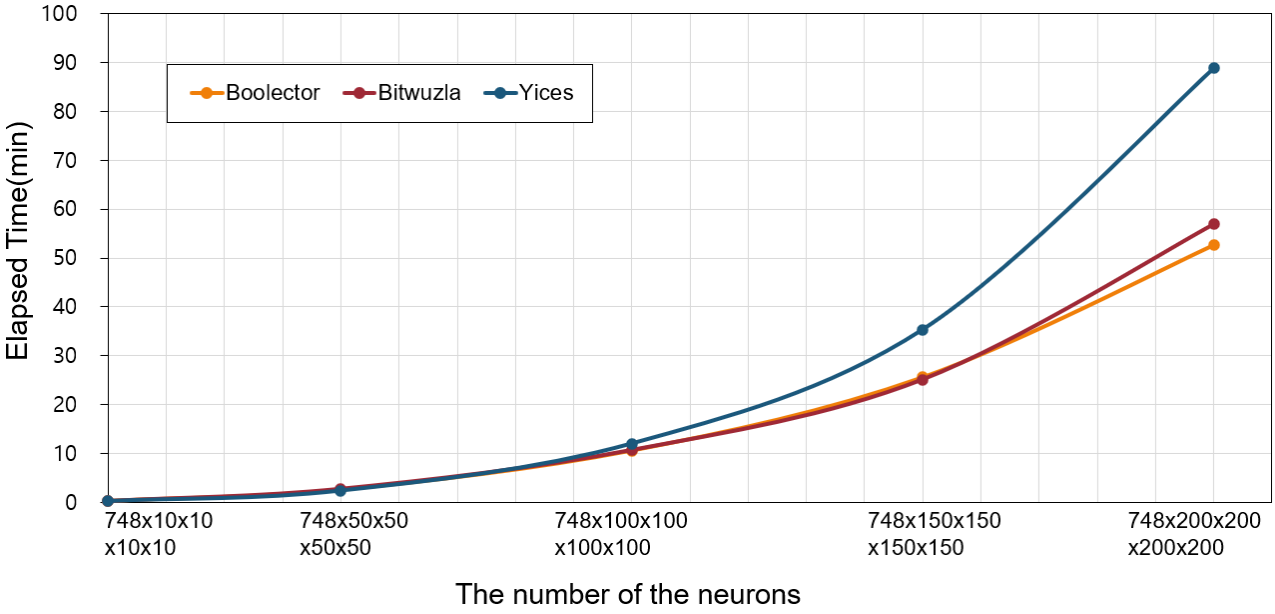

Fig. 26. The BMC program time

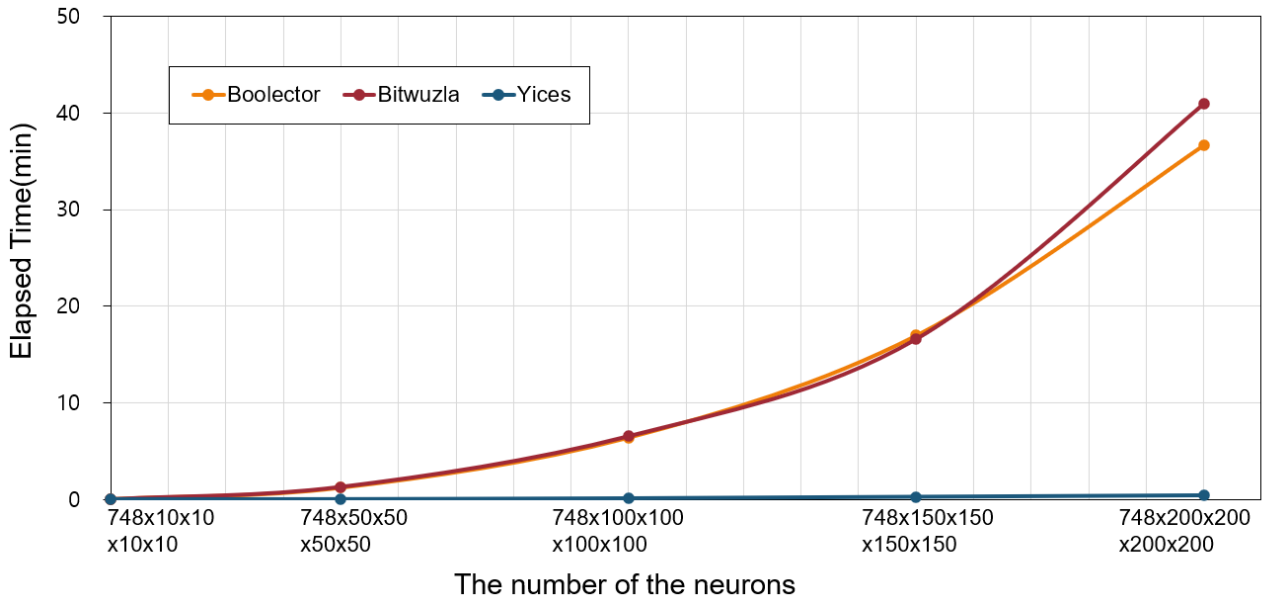

Fig. 27. The runtime decision procedure time

Fig 28 illustrates the encoding to solver time measured in the three solvers. The encoding to solver time of the Boolector or Bitwuzla solver is less than 2 minutes, and the proportion of the total verification time is less than 1%. However, the encoding to solver time of the Yices solver accounts for the largest proportion of verification time, and as the number of neurons rises, the ratio of encoding to solver time also rises sharply. It is a severe bottleneck for Yices that accounts for roughly 67% if there are more than 100 neurons and 76% if there are more than 150 neurons.

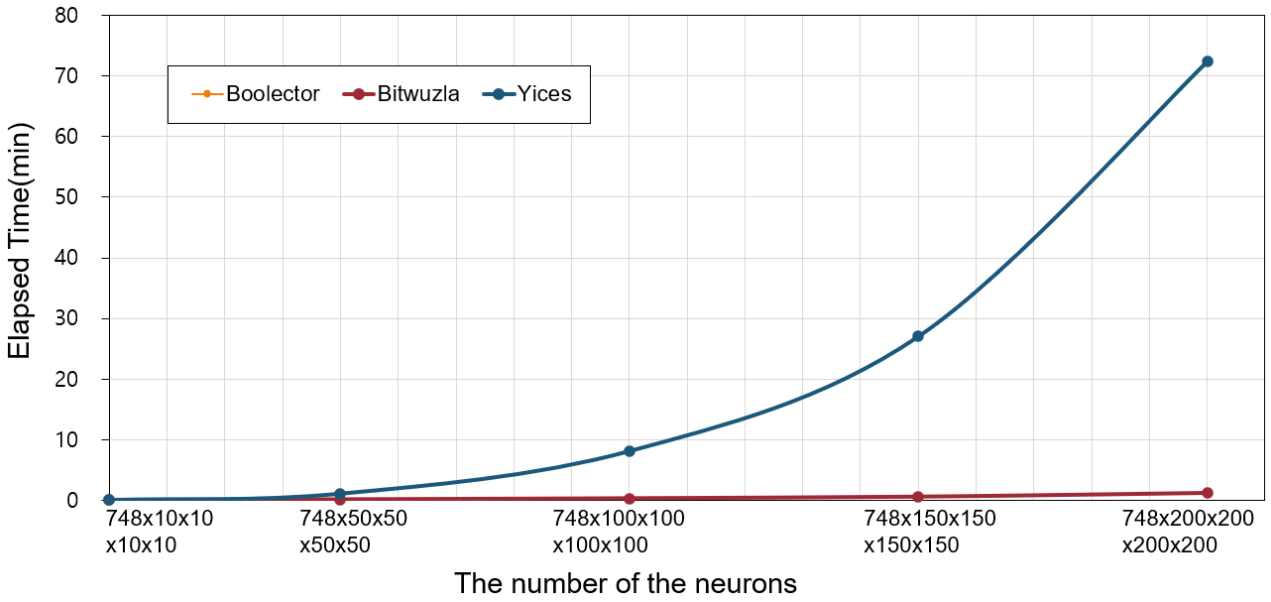

Fig. 28. The Encoding to solver time

4.6 7-layer BNNs

4.6.1 The structure of the 7-layer BNNs

The 7-layer BNN structure used for the test is shown in Figure 29. It consists of one input layer, five hidden layers, and one output layer, a total of seven layers. The input layer and the hidden layer are composed of binarisation, affine transformation, ReLU activation function, batch normalisation, and the output layer is composed of one ReLU activation function.

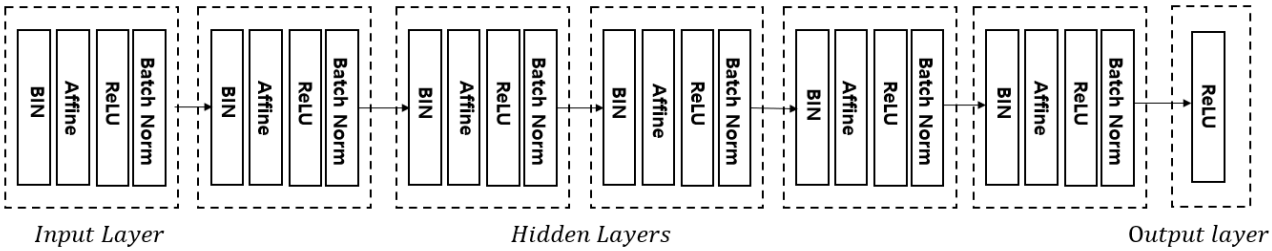

Fig. 29. The structure of the 7-layer BNN

4.6.2 The number of assignments

Fig 30 shows the number of assignments. The number of assignments of the 7-layer BNN increases more rapidly as the number of neurons in the hidden layer increases compared to 3-layer BNNs and 5-layer BNNs. The number of assignments grew by roughly 6.7 times when the number of neurons in the hidden layer increased fivefold from 10 to 50. When the number of neurons in the hid-

den layer tripled from 50 to 150, the number of assignments increased nearly fivefold. The verification time of the 7-layer BNN increases much faster than the 3-layer 3 BNN and the 5-layer BNN.

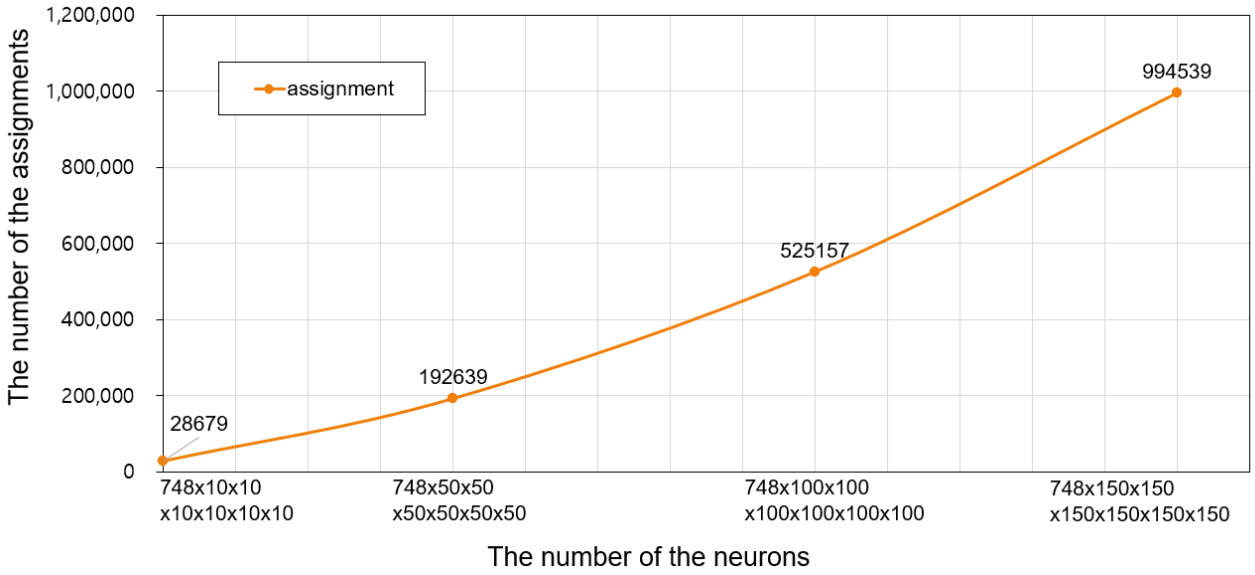

Fig. 30. The number of assignments

4.6.3 The maximum memory usage

Table 11 and Fig 31 illustrates the maximum memory usage. When verifying 7-layer BNNs, the Boolector or Bitwuzla solvers use a similar amount of memory, but the Yices solver requires more memory than Boolector or Bitwuzla. When the number of neurons is less than 100, the Yices solver needs approximately 20% more memory, and when the number is more than 150, it uses roughly 13% more memory. When the number of neurons doubled from 50 to 100, the memory usage of all solvers also increased by about 2.5 times. When the number of neurons tripled from 50 to 150, the average memory usage grew approximately 4.5 times. When verifying 7-layer BNNs, the average memory usage increases faster as the number of neurons increases. The most noteworthy is that all solvers already have more than 83% memory usage with 150 neurons. In particular, in the case of the Yices solver, it uses nearly 100% memory. We can assume that ESBMC got killed while verifying the 7-layer BNN with 200 neurons because of a lack of memory.

The number of neurons	Boolector (%)	Bitwuzla(%)	Yices(%)	Average(%)
748x10x10x10x10x10x10	0.230	0.034	0.015	4.09
748x50x50x50x50x50x50	3.057	1.273	0.136	19.64
748x100x100x100x100x100x100	14.689	8.132	0.491	49.62
748x150x150x150x150x150x150	46.592	30.158	1.025	88.37
748x200x200x200x200x200x200	killed	killed	killed	killed

Table 11. The maximum memory usage

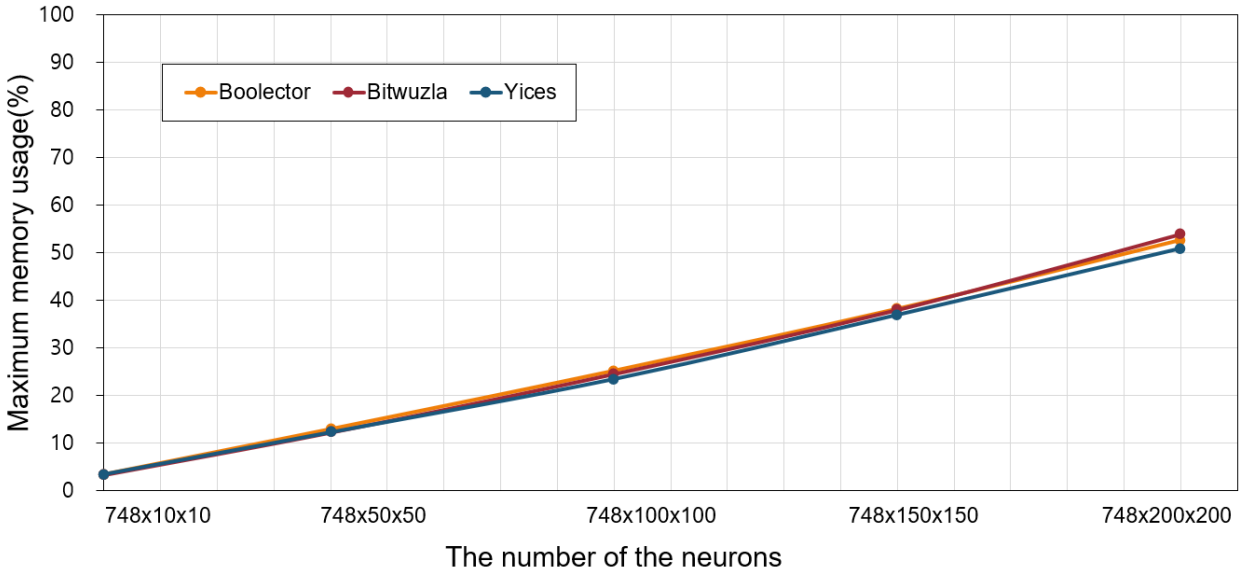

Fig. 31. The maximum memory usage

4.6.4 The verification time on Boolector

Table 12 and Figure 32 show BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Boolector solver.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10x10x10	0.230	0.034	0.015
748x50x50x50x50x50x50	3.057	1.273	0.136
748x100x100x100x100x100x100	14.689	8.132	0.491
748x150x150x150x150x150x150	46.592	30.158	1.025
748x200x200x200x200x200x200	killed	killed	killed

Table 12. The verification time on Boolector

The 7-layer BNN also exponentially increases the verification time as the number of neurons in the hidden layer rises. The number of neurons grew roughly tenfold from 10 to 100, while the verification time increased by 64 times, from 0.23 minutes to 14.689 minutes. Furthermore, when the number of neurons tripled from 50 to 150, the verification time rose approximately 15 times, from 3.057 to 46.592 minutes. When the number of neurons in the hidden layer was 200 in the 7-layer BNN, the verification failed because ESBMC was killed. Similar to the 3-layer BNN and 5-layer BNN, the main factor that occupies verification time is the runtime decision procedure and accounts for more than half if the number of neurons in the hidden layer is more than 100. In contrast, the portion of encoding to the solver time is trivial.

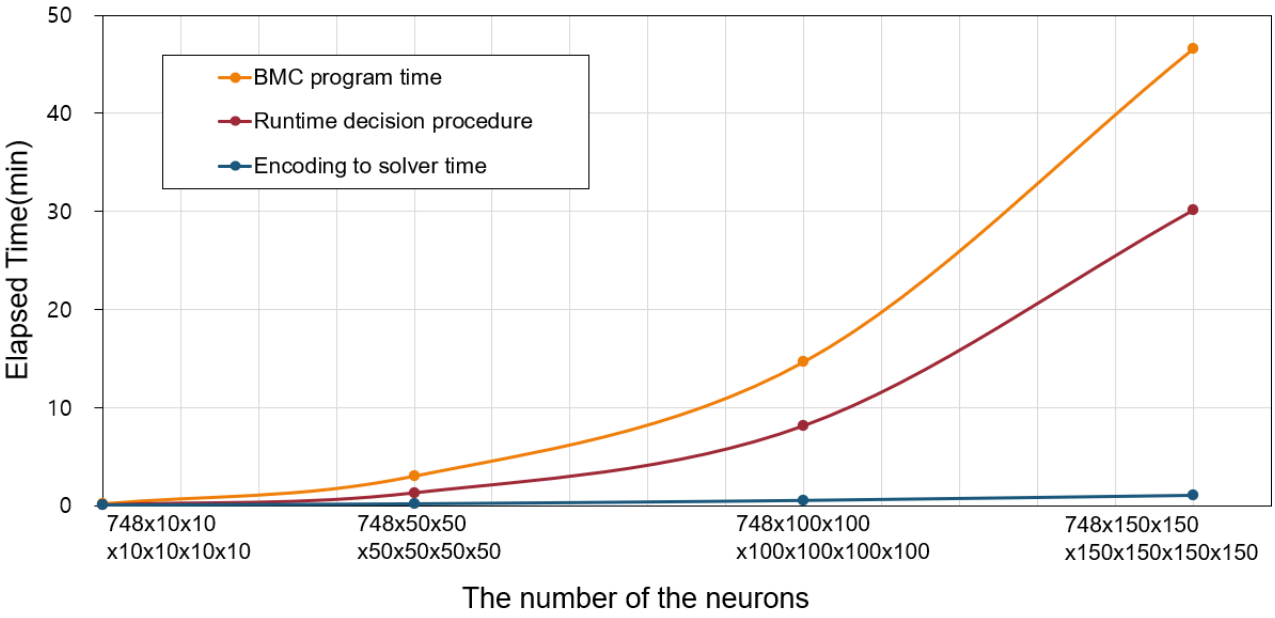

Fig. 32. The verification time on Boolector

4.6.5 The verification time on Bitwuzla

Table 13 and Figure 33 show the values of BMC program time, runtime decision procedure, and encoding to solver time, which are metrics measured by ESBMC while increasing the number of neurons on the Bitwuzla solver. The Bitwuzla solver has comparable metrics to the Boolector. The amount of time for runtime decision is considerably large, and the time taken for encoding to the solver time is negligible.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10x10x10	0.225	0.035	0.015
748x50x50x50x50x50	3.149	1.339	0.138
748x100x100x100x100x100	15.201	8.717	0.477
748x150x150x150x150x150	41.586	27.089	0.977
748x200x200x200x200x200	killed	killed	killed

Table 13. The verification time on Bitwuzla

The Bitwuzla solver also shows a similar tendency to the Boolector solver; as the total number of neurons increases, the verification time increases exponentially. When the number of neurons grew ten times from 10 to 100, the verification time increased 67.5 times from 0.225 minutes to about minutes. The verification time rose nearly 13 times as the neurons tripled from 50 to 150, from 3.149 to 41.586 minutes. The verification when the number of neurons in the hidden layer was 200 failed. The runtime decision procedure time is a significant factor that accounts for more than 57% of neurons from more than 100 neurons, and encoding to solver time has a minimal impact of less than 1 minute.

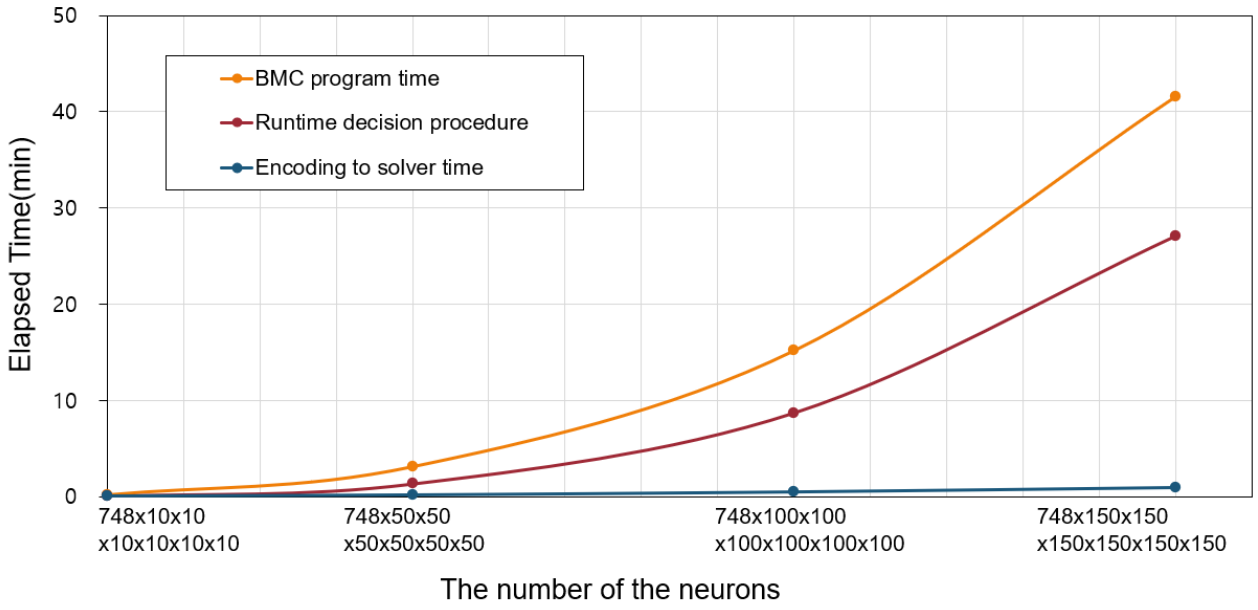

Fig. 33. The verification time on Bitwuzla

4.6.6 The verification time on Yices

Table 14 and Figure 34 illustrate the BMC program time, runtime decision procedure, and encoding to solver time assessed by ESBMC while increasing the number of neurons in the Yices solver. The Yices solver is the slowest of the solvers, followed by the Bitwuzla solver and Boolector. Unlike Boolector and Bitwuzla solvers, encoding to solver time is a substantial bottleneck, and the runtime decision procedure is quicker than encoding to solver time.

The number of neurons	BMC program time(Min)	Runtime decision procedure(Min)	Encoding to solver time (Min)
748x10x10x10x10x10x10	0.220	0.016	0.028
748x50x50x50x50x50x50	4.715	0.070	2.950
748x100x100x100x100x100x100	34.322	0.988	27.298
748x150x150x150x150x150x150	96.157	0.307	82.357
748x200x200x200x200x200x200	killed	killed	killed

Table 14. The verification time on Yices

The verification time of the Yices solver increases much more rapidly than other solvers as the number of neurons grows. The verification time grew nearly 59 times from 0.22 minutes to 34.322 minutes when the number of neurons was increased tenfold from 10 to 100. On the other hand, when the number of neurons tripled from 50 to 150, the verification time increased nearly 20 times, from 4.715 minutes to 96.157 minutes. The Booleter and Bitwuzla solvers increase in the same number of neurons compared to about 68 and 13 times, respectively. The Yices solver also failed to verify if the number of neurons in the hidden layer was more than 200. Unlike other solvers, the main bottleneck in Yices solver is encoding to solver time, and if the number of neurons is more than 100, it accounts for about 80% of Yice's total verification time. In contrast, runtime decision procedure time affects less than 1% of the total verification time.

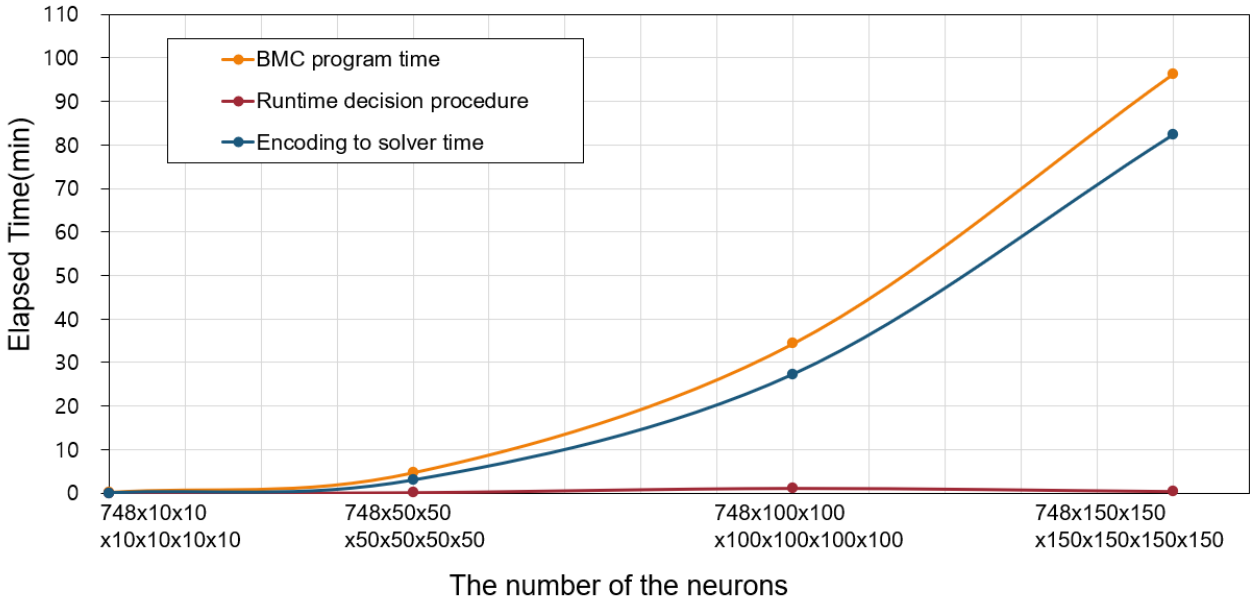

Fig. 34. The verification time on Yices

4.6.7 Comparison between the three solvers

Fig 35 shows the BMC program time measured in the three solvers. The verification times of the Boolector and Bitwuzla solver follow a similar pattern. When the number of neurons is less than 50, the Yices solver tends to be similar, but as the number of neurons increases, it is roughly twice as slow as other solvers. This gap is expected to widen as the number of neurons increases. Therefore, adopting the Boolector or Bitwuzla solver is more appropriate if the layer size is seven or higher.

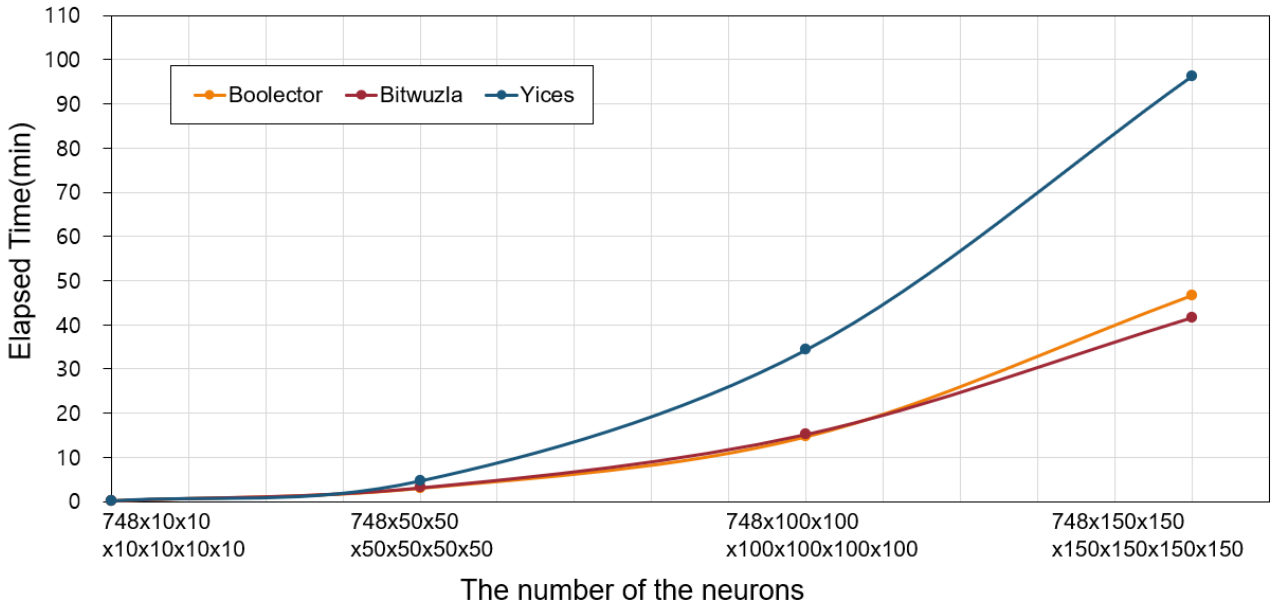

Fig. 35. The BMC program time

Fig 36 describes the runtime decision procedure time measured in the three solvers. The Boolector

and Bitwuzla solvers consume most of the total verification time during runtime decision procedure time. When the number of neurons exceeds 150, it accounts for more than 65% of the total verification time of the two solvers. In contrast, the runtime decision procedure time in the Yices solver only accounts for 1% of the total verification time, even though it tends to increase as the number of neurons in the hidden layer increases.

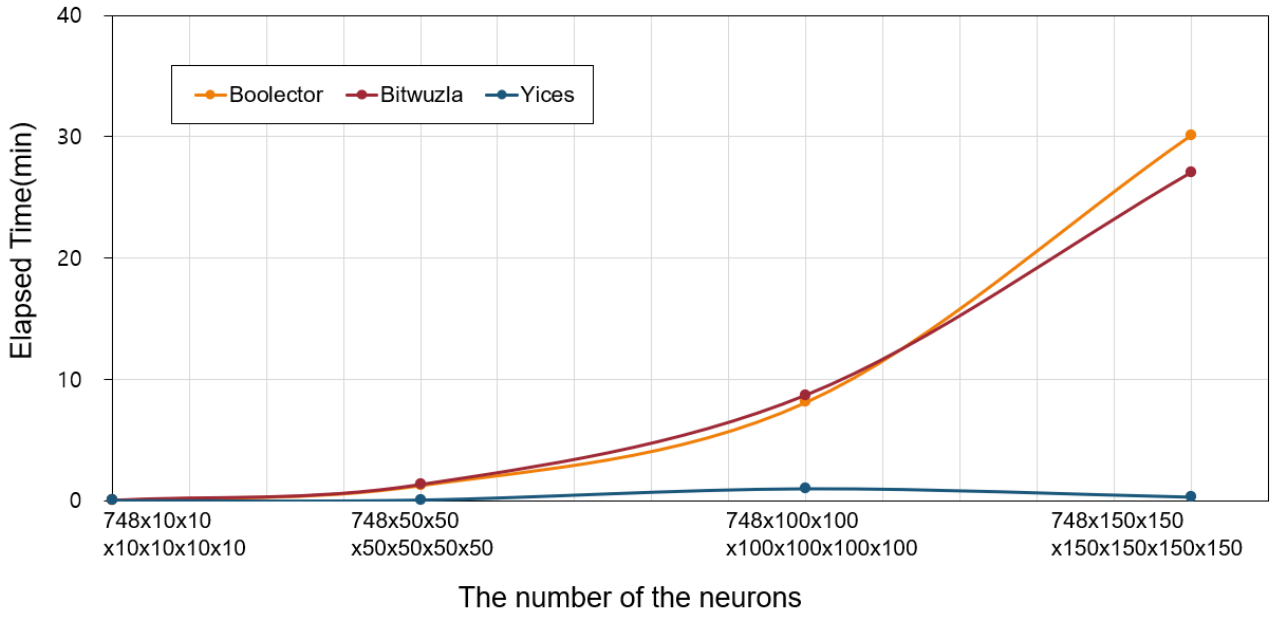

Fig. 36. The Runtime decision procedure time

Fig 37 illustrates the encoding to solver time measured in the three solvers. The encoding to solver time accounts for only a fraction of the Boolector and Bitwuzla verification time and does not account for 1% of the total verification time. However, the total verification time of the Yices solver is the most time-consuming factor, accounting for approximately 80% of the total verification time.

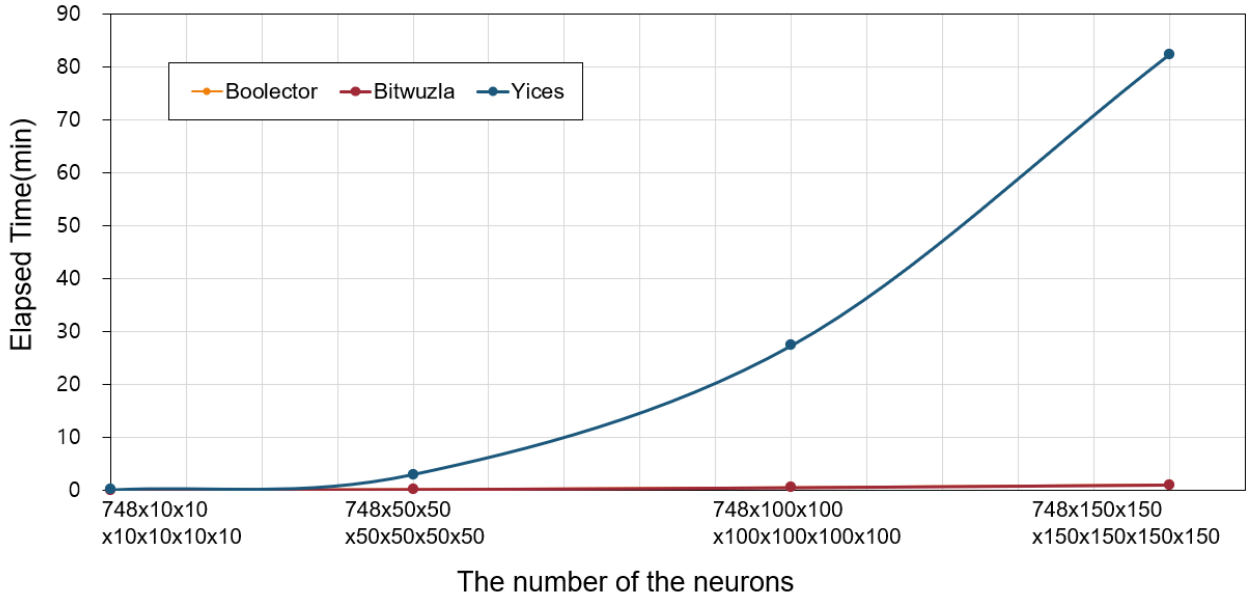

Fig. 37. The encoding to solver time

4.7 Comparison of the 3-layer, 5-layer, and 7-layer BNNs

We compare the verification metrics of the 3-layer, 5-layer, and 7-layer BNNs. We take the average verification time metrics of the three solvers: Boolector, Bitwuzla, and Yices.

4.7.1 Comparison of the number of assignments

Fig 38 shows the number of assignments. We can observe that as the number of layers increases, so does the number of assignments. The more layers there are, the faster the assignments grow, and the graph shape is expected to eventually shift from linear to exponential. The difference in the number of assignments is insignificant when the number of neurons in the hidden layer is 50 or less. However, the disparity widened when the number of neurons in the hidden layer exceeded 100, and it more than doubled when the number of neurons in the hidden layer exceeded 150.

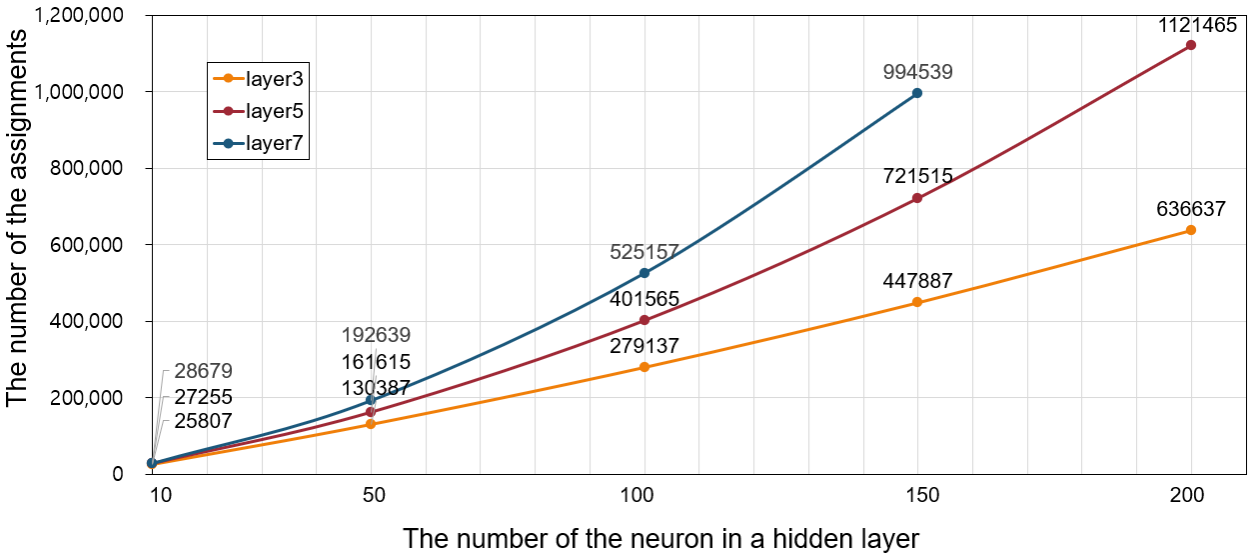

Fig. 38. The number of assignments

4.7.2 Comparison of the verification time

In Figure 39, the graph shows the average verification times of all solvers, including the Boolector, Bitwuzla and Yices. The BNN verification time has an exponential shape. As the layer size increases, it takes on the shape of a steeper exponential graph. When the number of neurons in the hidden layer is less than 50, there is an insignificant difference in verification time; however, when the layer size is larger than 100, there is a substantial difference in verification time. Specifically, when comparing the 3-layer BNN and 7-layer BNN, there was a difference of about twice when the number of neurons of the hidden layer was 50 but more than 3.5 times when the number of neurons of the hidden layer was 150. After that, if the number of neurons of the hidden layer increases further and verification becomes possible, the gap is expected to widen more greatly.

The hidden layer size	Layer 3(Min)	Layer 5(Min)	Layer 7(Min)
10	0.184	0.213	0.225
50	1.867	2.541	3.64
100	6.949	11.08	21.404
150	17.364	28.696	61.445
200	27.011	66.163	Killed

Table 15. The verification time

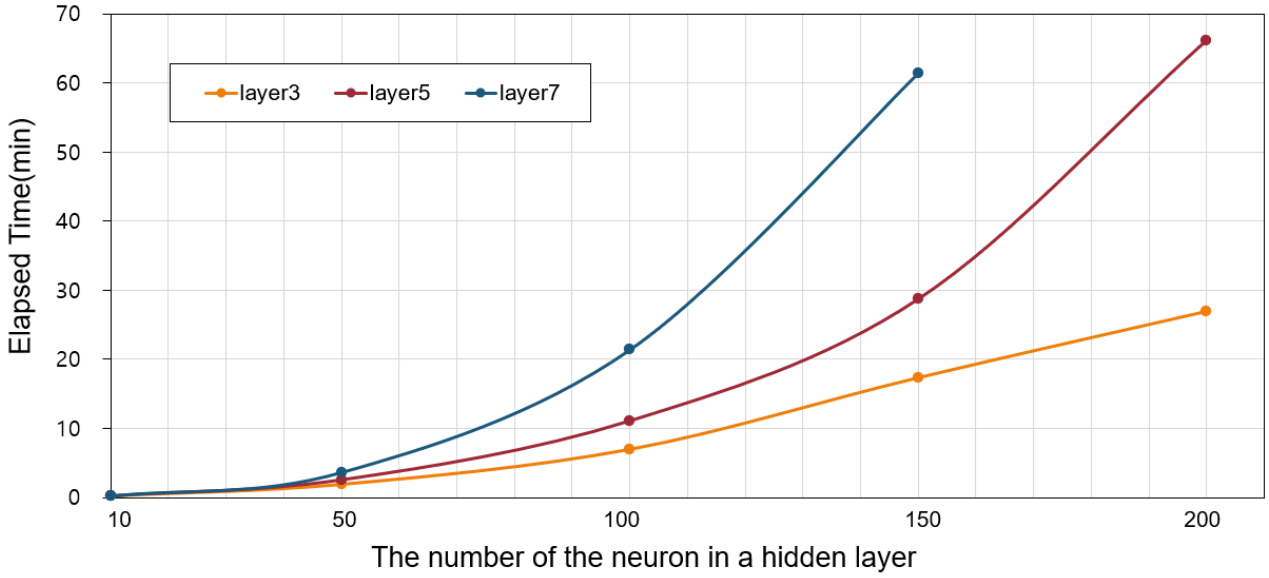

Fig. 39. The verification time

4.7.3 Comparison of the maximum memory usage

Figure 40 and Table 16 illustrates the maximum memory usage graph of the 3-layer, 5-layer, and 7-layer BNNs. Generally, as the number of layers and neurons increases, the maximum memory usage rapidly grows. The 7-layer BNN shows the steepest gradient compared to others. When the number of neurons was 50 or less, the verification time for all layers was relatively similar. When the number of neurons increased to 100, 7-layer BNNs needed almost 1.6 times more memory than 3-layer BNNs. When the number of neurons was 200, the difference between the two numbers was more than twice.

Notably, ESBMC got killed while verifying the 7-layer BNNs with 200 neurons in the hidden layer. We can infer that it is due to insufficient memory. The maximum memory usage required for the verification was 93.33% for the 5-layer BNN with 200 neurons. In 5-layer BNNs, the average maximum memory usage increased by about 1.46 times when the number of neurons increased from 150 to 200. Applying this to 7-layer BNNs, we can infer that it requires nearly 130% of memory when the number of hidden layer neurons is 200. Currently, we uses 12G memory; therefore, a minimum of 16G of memory is required to verify a 7-layer BNN with 200 neurons. The verification is likely to succeed in this environment. It is expected that more memory will be needed to manage more sized neurons. Given this trend, as the number of neurons increases, the difference in the

maximum memory usage required between each layer is expected to increase.

The hidden layer size	Layer 3(%)	Layer 5(%)	Layer 7(%)
10	3.39	3.68	4.09
50	12.47	16.06	19.64
100	24.36	37.04	49.62
150	37.68	63.79	88.37
200	52.45	93.33	Killed

Table 16. The maximum memory usage

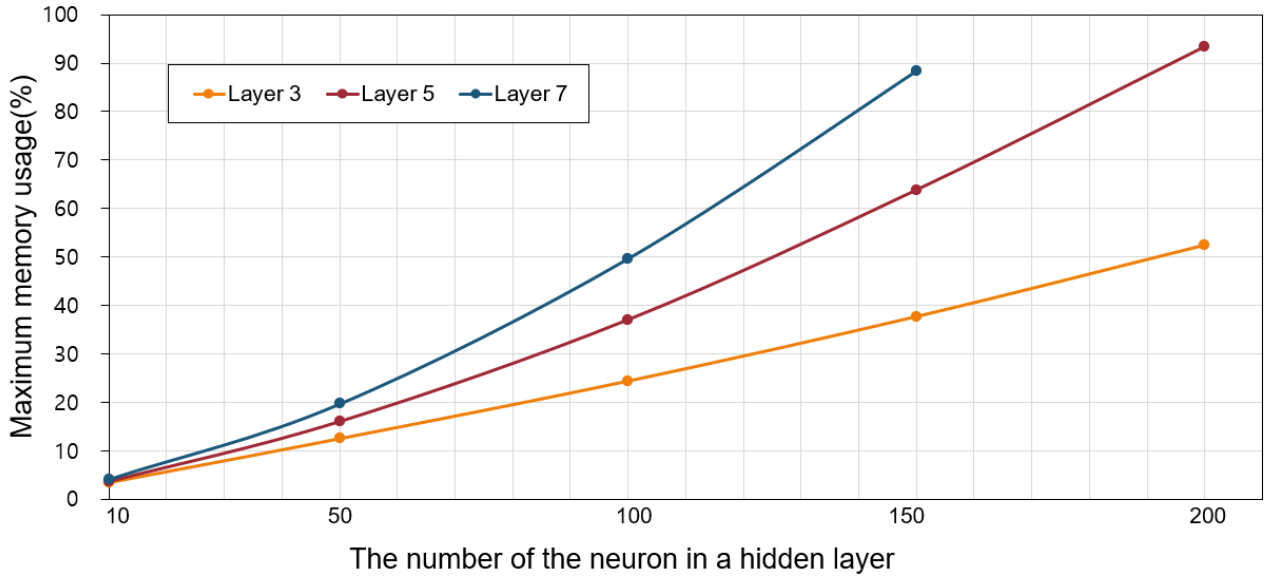

Fig. 40. The maximum memory usage

4.8 Summary

In the results and discussion, we explain the results of the verification experiments for the 3-layer, 5-layer and 7-layer BNNs. The number of hidden layer neurons increases to10, 50, 100, 150, and 200, with the input data size fixed at 28*28. The verification is performed on ESBMC with the Boolector, Bitwuzla, and Yices solvers. We conclude that as the number of layers and neurons grows, so does the verification time, and the Boolector and Bitwuzla solver handle scalability better than the Yices solver. The Boolector is the fastest among them, even though the difference in time verifying BNNs between the Boolector and Bitwuzla is trivial.

5 Conclusions and future work

5.1 Conclusions

BNN is a sort of ANN that uses binary parameters at runtime to improve performance and reduce computing resource consumption. Due to these advantages, BNN has recently received many expectations that it can play an essential role in the universal use of AI. However, verifying BNNs while managing the scalability is challenging in current ways. This study is conducted to verify these BNNs more efficiently.

This study proposes BNN training and testing methods optimised for ESBMC. We discuss different strategies for optimising code, such as conversion data type to boolean data type, lookup tables, and macro functions. As a result, we confirm through experiments that medium-sized BNN verification is possible within a reasonable time through ESBMC. We also demonstrate that the verification time increases exponentially as the number of layers and neurons increased.

In addition, we also compare the maximum memory usage and ESBMC metrics by conducting experiments with the Boolector, Bitwuzla, and Yices solvers. As the number of neurons and layers raises, the maximum memory usage also increases. In 3-layer BNN and 5-layer BNN, memory usage between solvers is similar, but in 7-layer BNN, the Yices solver requires more memory than the Boolector or Bitwuzla solvers. With respect to the ESBMC metrics, we confirm that the Boolector and Bitwuzla show almost similar performance and characteristics, but overall, the Boolector is slightly faster than the Bitwuzla. The Yices solver has distinctly different features from the Boolector and Bitwuzla solvers. In particular, the main bottleneck of the Yices solver is the encoding to solver time, unlike other solvers whose main bottleneck is runtime decision procedure. The Yices solver is faster than Boolector and Bitwuzla solvers when there are three or fewer layers, but performance drops sharply, and scalability decreases when there are more than five layers due to the massive requirement of memory. Therefore, the Boolector or Bitwuzla solver is more appropriate for deep learning than the Yices solver.

5.2 Future work

We conduct experiments focusing on the metric measured during successful verification. There is a need for new studies focusing on adversarial attack verification that does not satisfy safety characteristics is needed. If the verification fails, the ESBMC provides additional information on the cause of the failure and takes longer to verify.

In addition, the experiments can verify medium-sized BNNs but fail to verify large-sized BNNs. We suggest conducting new experiments in an environment where the memory is larger than 12G.
If the number of neurons in a 5-layer BNN is more than 200, the maximum memory usage for all

solvers is approximately 92%, so it is expected that the memory usage of the 7-layer BNN with 200 neurons exceeded 100%. Therefore, it is highly likely that the ESBMC was killed due to a lack of memory during execution. Therefore, we propose to perform new experiments on a computer with at least 20G of memory.

We also suggest that it is essential to devise optimisation algorithms for verifying deep neural networks with thousands of hidden layer sizes. It is worth considering introducing faster solvers and conducting experiments because a solver occupies most of the verification time. Balunovic et al. proposed a novel ”learning to solve SMT formula” method [33]. Their strategy achieved up to 100x runtime improvement over a state-of-the-art SMT solver by defining the problem of solving SMT formulae as a tree search problem[33]. In addition, shift-based batch normalisation suggested by Courbariaux et al. can be applied for the testing phase [13]. Since shift-based batch normalisation substitute multiplications and divisions with shift operators, it can guarantee quicker verification. [13].

In addition, if the accuracy of the BNN is critical, it is possible to minimise the binarisation applied to the parameter. The currently proposed method binarises activations, weights, and biases. However, we can use real-valued biases or activations to reach higher accuracy. In addition, we can tune hyperparameters such as the number of epochs, batch size and optimisers. Since this experiment is optimised with an Adam optimiser, other more advanced optimisers, such as Nadam or Angular Grad, can be employed. In addition, this study limited the activation function to ReLU to improve the verification speed. However, higher accuracy can be achieved by replacing ReLU with Tanh for the hidden layers and softmax for the output layer. Finally, we use fully-connected BNNs, which can lead to overfitting and thus reduce accuracy. Therefore, we believe that accuracy can be improved by implementing partially-connected BNNs with dropout.

References

[1] L. Sena, X. Song, E. Alves, I. Bessa, E. Manino, L. Cordeiro, *et al.*, “Verifying quantized neural networks using smt-based model checking,” *arXiv preprint arXiv:2106.05997*, 2021.

[2] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” *Advances in neural information processing systems*, vol. 28, 2015.

[3] T. Simons and D.-J. Lee, “A review of binarized neural networks,” *Electronics*, vol. 8, no. 6, p. 661, 2019.

[4] K. Jia and M. Rinard, “Efficient exact verification of binarized neural networks,” *Advances in neural information processing systems*, vol. 33, pp. 1782–1795, 2020.

[5] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, “Backpropagation and the brain,” *Nature Reviews Neuroscience*, vol. 21, no. 6, pp. 335–346, 2020.

[6] S. Sharma, S. Sharma, and A. Athaiya, “Activation functions in neural networks,” *towards data science*, vol. 6, no. 12, pp. 310–316, 2017.

[7] Z. Qiumei, T. Dan, and W. Fenghua, “Improved convolutional neural network based on fast exponentially linear unit activation function,” *Ieee Access*, vol. 7, pp. 151 359–151 367, 2019.

[8] M. A. Mercioni and S. Holban, “The most used activation functions: Classic versus current,” in *2020 International Conference on Development and Application Systems (DAS)*, IEEE, 2020, pp. 141–145.

[9] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2017.

[10] M. D. Zeiler, “Adadelta: An adaptive learning rate method,” *arXiv preprint arXiv:1212.5701*, 2012.

[11] R. Zaheer and H. Shaziya, “A study of the optimization algorithms in deep learning,” in *2019 third international conference on inventive systems and control (ICISC)*, IEEE, 2019, pp. 536–539.

[12] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, “Understanding batch normalization,” *Advances in neural information processing systems*, vol. 31, 2018.

[13] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.

[14] G. Hinton, L. Deng, D. Yu, *et al.*, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[15] D. Beyer, “Status report on software verification,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2014, pp. 373–388.

[16] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking.,” *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[17] W. Gong and X. Zhou, “A survey of sat solver,” in *AIP Conference Proceedings*, AIP Publishing LLC, vol. 1836, 2017, p. 020 059.

[18] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of model checking*, Springer, 2018, pp. 305–343.

[19] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, “Efficient formal safety analysis of neural networks,” *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[20] B. Alpern and F. B. Schneider, “Recognizing safety and liveness,” *Distributed computing*, vol. 2, no. 3, pp. 117–126, 1987.

[21] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, “Esbmc 5.0,” 2018.

[22] R. Brummayer and A. Biere, “Boolector: An efficient smt solver for bit-vectors and arrays,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2009, pp. 174–177.

[23] B. Dutertre, “Yices 2.2,” in *International Conference on Computer Aided Verification*, Springer, 2014, pp. 737–744.

[24] L. d. Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[25] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, “The mathsat5 smt solver,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2013, pp. 93–107.

[26] C. Barrett, C. L. Conway, M. Deters, *et al.*, “Cvc4,” in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 171–177.

[27] A. Stump, C. W. Barrett, and D. L. Dill, “Cvc: A cooperating validity checker,” in *International Conference on Computer Aided Verification*, Springer, 2002, pp. 500–504.

[28] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, “Verifying properties of binarized deep neural networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[29] A. Shih, A. Darwiche, and A. Choi, “Verifying binarized neural networks by angluin-style learning,” in *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2019, pp. 354–370.

[30] G. Amir, H. Wu, C. Barrett, and G. Katz, “An smt-based approach for verifying binarized neural networks,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2021, pp. 203–222.

[31] G. Kovásznai, K. Gajdár, and N. Narodytska, “Portfolio solver for verifying binarized neural networks,” in *Annales Mathematicae et Informaticae*, Eszterházy Károly Egyetem Líceum Kiadó, vol. 53, 2021, pp. 183–200.

[32] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[33] M. Balunovic, P. Bielik, and M. Vechev, “Learning to solve smt formulas,” *Advances in Neural Information Processing Systems*, vol. 31, 2018.