Verifying Quantized 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
2023

Student ID Number
11052209

The School of Engineering

Contents

Word Count: 12350

Abstract

As the complex problems that confront neural networks increase, the computational efficiency of networks decreases, accompanied by a surge in required memory capacity. This convergence of problems renders the application of neural networks to embedded or mobile devices a formidable challenge. To ameliorate this, the strategy of quantization is employed, albeit with introducing quantization errors that undermine the neural network's safety. Consequently, the meticulous verification of quantized neural networks assumes paramount significance. In the present study, the verification of quantized neural networks is implemented through the SMT-based model checking. Concretely, the neural network model is expressed using the C programming language, wherein floating-point numbers convert to fixed-point numbers, resulting the quantization process. Subsequently, Efficient SMT-based Bounded Model Checker (ESBMC) is employed to verify quantized neural networks. The small-size neural networks in this study can be subjected to verification within a span of ten minutes. It is noteworthy that different levels of quantization exert discernible influences on the verification process, encompassing both time and outcomes. Reducing quantization levels has the potential to diminish the time required for verification processes. However, this alteration might engender notable disparities between verification outcomes and the non-quantized results. Conversely, opting for elevated quantization levels generally renders increased verification time, yet affords outcomes which are similar to the non-quantized results.

Declaration

I confirm that this dissertation is my 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 dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rightsin 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 dissertation, 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 entered into. 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 worksin the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, 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 dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy, inany relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations.

1. Introduction

Currently, neural networks have found application in safety-critical domains, including but not limited to medical diagnostics and autonomous vehicular navigation. Within these domains, even minor errors can precipitate significant ramifications. For instance, in the context of autonomous driving, an erroneous judgement holds the potential to incite vehicular accidents or loss of human life. It is worth noting, however, that neural networks inherently exhibit opacity, impeding the explication of their internal processes. Furthermore, their susceptibility to slight deviations within input data engenders instability in outcomes, rendering them susceptible to adversarial incursions. Consequently, meticulous verification of neural networks assumes paramount importance within safety-critical realms.

The prevalence of deep learning models and neural networks in diverse applications has concurrently led to substantial growth in computational complexity and memory demands. Given the mounting necessity for efficient neural network deployment, particularly within resource-limited settings such as mobile devices, the quantization of neural networks has garnered considerable attention. Quantization methodologies strive to address these challenges by diminishing model size and computational requisites while maintaining minimal accuracy degradation. However, the procedure of quantization may lead to a multitude of issues, such as approximation inaccuracies and misjudgment. These challenges have the potential to exacerbate the existing security concerns associated with artificial neural networks, thereby diminishing the robustness, dependability, and reliability of quantized models. Consequently, the verification of quantized neural network have gained paramount importance. In this research, the project will focus on quantizing artificial neural networks and verifying their safety through the employment of Satisfiability Modulo Theories (SMT) based model checking techniques. The examination encompasses an in-depth analysis of the repercussions arising from various quantization levels on the verification process.

This paper is structured into five primary sections. The initial segment acquaints the reader with the research's focal issues and underlying motivations. The subsequent section provides an exposition on the background relevant to this undertaking, encompassing Artificial Neural Networks (ANNs), Quantized Neural Networks (QNNs), Satisfiability Modulo Theories (SMT), safety properties, and existing verification methodologies. The third section elucidates the specific approach employed in conducting the experimentation, encompassing the realization of ANNs, quantization techniques for neural networks, and the process of QNN verification. The fourth section expounds upon the datasets and environment employed within the experiment. The pertinent experimental outcomes are shown, subsequently scrutinizing and deliberating upon the ramifications stemming from distinct quantization levels, as inferred from the experiment's findings. Section V serves to conclude the project's accomplishments, then describing prospective avenues for further research and improvement.

2. Background

2.1. Artificial Neural Networks (ANNs)

ANNs are a type of computational model inspired by the human brain. McCulloch and Pitts [1] proposed a binary logic gate as the idea of a neuron in neural networks, which could be used to perform simple logical calculus in neural networks. In the late 1950s, Rosenblatt [2] introduced a probabilistic model called perceptron, which is a type of neural network that can be trained to perform simple classification tasks. In the later years, many researchers have proposed different types of neural networks and machine learning algorithms to handle hard and complex tasks, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Reinforcement learning (RL), Generative Adversarial Networks (GAN), and Long Short Term Memory (LSTM).

Intended to mimic the structure and function of the human brain, ANNs consist of a large number of computational units called neurons, and neurons are connected by directed and weighted connections. Figure 1 shows the network structure of a simple ANN. In general, ANNs have multiple layers, which are input layers, hidden layers, and output layers, and each layer consists of a set of neurons. The input layer is the first layer of ANNs, which receives and passes the data to the next layer. The hidden layers are the essential part of ANNs, which take inputs from the previous layer and produce outputs to the next layer. During the process of hidden layers, ANNs model and learn complex relationships and features in the input data by adjusting weights. The output layer is the final layer of ANNs, which produces the output or prediction of the network. The output layer transforms the data from the previous hidden layer into the task-specific form.

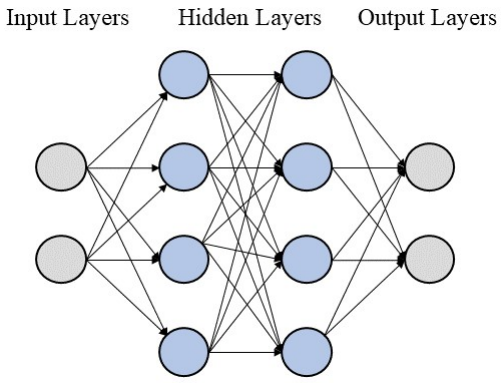

Figure 1: The network structure of a simple ANN

Figure 2 shows the workflow of a single neuron. The neuron receives a set of inputs, and each input is multiplied by a weight which determines the strength of the connection between neurons. Then, the neuron calculates the sum of the weighted inputs and applies an activation function to the result.

Generally, activation functions are non-linear functions, which introduce non-linear factors to neural networks and assist networks in understanding and learning complex relationships. Finally, the neuron passes the result of the activation function as the output to other neurons in the network or to the output layer.

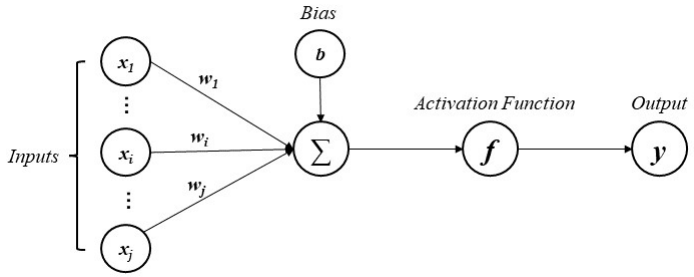

Figure 2: The workflow of a single neuron

The artificial neural network fundamentally comprises two primary algorithms: forward propagation and back propagation. Forward propagation involves the processing of input data through successive layers until the final output is attained. A fundamental aspect of forward propagation lies in the utilization of activation functions. These activation functions play a pivotal role in determining the output of each layer during the forward propagation process. The process of back propagation entails the comparison of the output obtained through forward propagation with the actual result, leading to the computation of the loss. Following this, the weights and biases of each neuron are iteratively adjusted in accordance with this loss. The central significance of back propagation resides in the careful selection of the optimization algorithm. The selection of an appropriate optimizer and loss function assumes a pivotal role in expediting convergence and elevating the overall learning efficiency of the neural network.

2.1.1. Activation functions. The activation function determines the activation states of neurons within a neural network. It assumes the form of a nonlinear function that introduces non-linearity into the network, thereby facilitating the learning of intricate patterns and features from the data. Several activation functions are commonly employed, including Sigmoid, TanH, Rectified Linear Unit (ReLU), and softmax.

1) Sigmoid function The sigmoid function represents a widely employed mathematical function renowned for its characteristic S-shaped curve in space. Its mathematical expression is defined as follows:

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

where 'x' signifies the input to the neuron, and 'e' denotes the base of the natural logarithm. This function yields output values within the range of 0 to 1, with values approximating 1 as 'x' tends toward positive infinity and values approaching 0 as 'x' tends toward negative infinity. The sigmoid

function finds prominent utility in models that produce predicted probabilities as output because of its value range. Despite its extensive usage, the sigmoid function is not without limitations. When the input takes on large or small values, the gradient of the function tends to approach 0, resulting in the problem of gradient vanishing. Moreover, in the vicinity of 0, the sigmoid function exhibits a substantial gradient, while further away from 0, the gradient diminishes, potentially leading to the problem of gradient explosion. Additionally, the function's output is not centered on 0, which can impede the efficiency of weight updates and adversely impact the optimization process [3]. Another concern is that the sigmoid function mainly involves exponential calculations, which can impede computational efficiency, resulting in slower execution on computers.

2) Tanh function The hyperbolic tangent function, known as Tanh, shares similarities with the sigmoid function, albeit with some distinctions. Its mathematical expression is as follows:

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

Figure 3 shows the images of sigmoid and Tanh functions, where the blue curve is the sigmoid function, and the red curve is the Tanh function. It is evident that the Tanh also exhibits an S-shaped curve, but with a wider value range (-1, 1), rendering it similar to a re-scaled variant of the sigmoid function. The

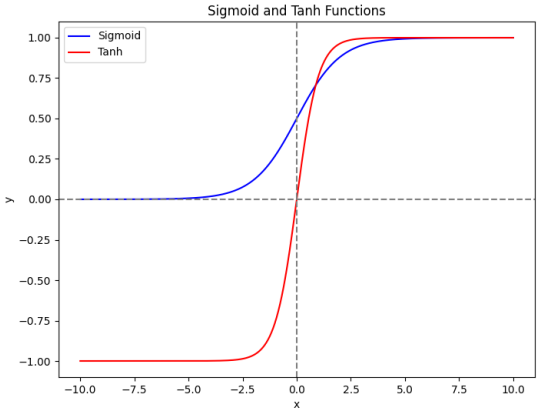

Figure 3: Sigmoid and Tanh functions

Tanh function yields output values within the range of -1 to 1. For input values approaching negative infinity, the output tends to -1; conversely, for input values approaching positive infinity, the output tends to 1. Additionally, when the input is 0, the output is 0, thereby centering the output values around 0 and enhancing optimization efficiency [3]. Compared to the sigmoid function, Tanh exhibits a larger gradient in proximity to the origin, effectively mitigating the issue of gradient vanishing. However, it does not entirely resolve the gradient vanishing problem, as for large or small input values, the function's gradient still tends to 0, resulting in the same challenge of gradient disappearance [4]. Furthermore, similar to the sigmoid function, Tanh also primarily relies on exponential operations, leading to relatively high computational costs.

3) ReLU function The Rectified Linear Unit (ReLU) function is a widely utilized activation function in the realm of deep learning. This function is mathematically defined as follows:

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

Essentially, when the input value is greater than or equal to 0, the output value remains the same as the input, whereas for input values less than 0, the output value becomes 0. In comparison to previously introduced activation functions 'sigmoid' and 'Tanh', ReLU exhibits distinct advantages. Notably, when the input of the ReLU function is positive, the gradient remains consistently equal to 1. This attribute enables ReLU to circumvent the issue of gradient saturation. Additionally, ReLU exhibits faster computational efficiency due to its linear nature, fostering accelerated processing in deep learning networks [5]. However, despite these advantages, the ReLU function also presents some drawbacks. Specifically, it may lead to the occurrence of the 'Dead ReLU Problem'. During the training process, certain neurons may become inactive and never be activated, granting ReLU an advantage of network sparsity [3]. Nonetheless, a large number of neurons are dead, posing challenges in neural network training and hindering the performance [3]. Furthermore, the output of the ReLU function is either 0 or a positive value, thereby indicating that the ReLU function is not centered at 0.

4) ReLU variants In addressing the issue concerning the ReLU function, several scholars have put forth a number of variants, such as Leaky ReLU [6], Exponential Linear Unit (ELU) [7], and Parametric ReLU (PReLU) [8]. Figure 4 shows the images of these functions.

The Leaky ReLU is defined as:

$$\text{Leaky ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha \cdot x, & \text{if } x < 0 \end{cases} \tag{4}$$

The Leaky ReLU introduces a slope, denoted by α, when the input value is less than 0. The slope α is a small real number between 0 and 1. By incorporating this slope, Leaky ReLU effectively mitigates the "Dead ReLU Problem", as it produces non-zero outputs for negative input values [6]. Moreover, the function mitigates the problem of gradient disappearance, as neurons with inputs less than zero can be updated during the training process.

The ELU also presents a resolution to the issues associated with the ReLU. Its mathematical expression is defined as follows:

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha \cdot (e^x - 1), & \text{if } x < 0 \end{cases} \tag{5}$$

The slope parameter α in the ELU bears resemblance to Leaky ReLU, as it is a small value greater than 0 yet less than 1. However, a key distinction lies in the ELU function's possession of a smooth curve within the negative value range [7]. This attribute has advantageous in mitigating the issue of gradient

explosion and promoting faster convergence during the training process. Additionally, ELU facilitates the average of activations around zero, consequently fostering enhanced learning capability within the neural network [7].

The mathematical formulation of PReLU is similarity to that of Leaky ReLU; however, a notable distinction lies in the fact that PReLU incorporates a learnable slope parameter 'a,' as opposed to a fixed value in Leaky ReLU. Empirical investigations have demonstrated the remarkable efficacy of PReLU in handling extensive image recognition tasks of substantial scale [8].

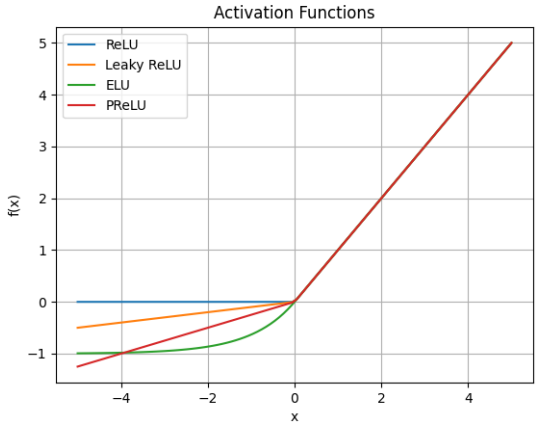

Figure 4: ReLU and its variants

5) Softmax function The Softmax function represents an activation function widely employed in multi-classification tasks and is frequently adopted as the activation function for the output layer. The Softmax function is utilized to calculate the probability distribution of each element within the input array. This distribution assigns a probability value to each element, falling within the range of 0 to 1, and collectively sums to 1 across all elements. To illustrate, consider an input vector such as [1.5, 2.7, 3.4, 2.3]. Once subjected to the Softmax function, the resulting output vector becomes [0.08, 0.25, 0.5, 0.17]. Consequently, the model employs this probability distribution to make predictions, selecting the class with the highest probability as the predicted value; thus, in this instance, designating the third class as the predicted outcome.

In this project, the activation function employed in the hidden layer is ReLU due to its ease of quantification and verification. Additionally, for the output layer, the softmax function is utilized, given that the project predominantly involves multi-classification tasks.

2.1.2. Optimization algorithm. The optimization algorithm within the neural network iteratively refines the parameters during the course of model training, aiming to discern the optimal solution of the objective function or to minimize the value of the loss function. Prevalent optimization techniques encompass the gradient descent method, the momentum optimization method, and the adaptive learning rate optimization method.

1) Gradient descent The gradient descent algorithm stands as a preeminent and extensively employed optimization technique within the realm of neural network model training. Virtually all contemporary libraries dedicated to deep learning boast an array of meticulously crafted implementations encompassing diverse and refined gradient descent algorithms [9]. The crux of the gradient descent methodology resides in its capacity to solve the minimal value along the gradient descent trajectory. Effecting a reduction in the incurred loss mandates the incremental advancement of parameters along a trajectory that is diametrically opposed to the gradient orientation [9]. Assuming the model parameters are designated as θ, the loss function can be denoted as $J(\theta)$. The symbol $\nabla J(\theta)$ corresponds to the partial derivative of the loss function $J(\theta)$ concerning the parameters θ, constituting the gradient. Additionally, α represents the learning rate, determining the magnitude of each parameter update iteration. Then, the mathematical expression is as follows: $\theta_{i+1} = \theta_i - \alpha \cdot \nabla J(\theta_i)$. Based on the volume of sample data employed during each parameter update, the gradient descent methodology can be categorized into three distinct variants: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-batch Gradient Descent [9]. BGD utilizes the entirety of samples within the training set to compute the gradient on each occasion an iterative parameter update is executed. The gradient descent procedure employed in BGD ensures that each parameter update's direction aligns with the global average gradient. Consequently, this method possesses the capability to guarantee the attainment of the global optimal solution or, equivalently, the global minimum for error surfaces that exhibit convex characteristics [9]. Nonetheless, it is imperative to note that the comprehensive inclusion of all samples, particularly when confronted with voluminous datasets, engenders a protracted pace of convergence and imparts a substantial burden upon computational resources. Diverging from the method employed in BGD, SGD adopts an approach by stochastically selecting one sample from the training set during each parameter update, over which it subsequently computes the corresponding gradient. The inherent advantage of this stochastic approach resides in its notable expedited training speed. This swifter speed holds true even when confronted with a large amount of samples, for the process necessitates a mere fraction of the entire dataset to advance towards the optimal solution. However, it is imperative to acknowledge that owing to its random nature, each iterative step within the SGD does not invariably align with the overarching optimization direction. Consequently, consequential fluctuations manifest within the gradient descent procedure, potentially engendering susceptibility to local optima entrapment or vacillations between disparate local optima [9]. Furthermore, SGD displays a marked sensitivity to the chosen learning rate. In instances where the learning rate is small, the progression towards convergence tends to be slowly characterized. Conversely, large learning rates tend to engender erratic fluctuations during convergence. Mini-batch Gradient Descent amalgamates the principles of BGD and SGD, wherein it selects a subset of data, known as a mini-batch, for each parameter update to compute its corresponding gradient. By synthesizing the advantages in both preceding methodologies, Mini-batch Gradient Descent ensures a dual advantage: it upholds training efficiency while simultaneously securing the accuracy of ultimate

convergence outcomes.

2) Momentum The Momentum optimization algorithm incorporates the principle of momentum from the field of physics into the context of SGD [10]. In essence, in the absence of opposing forces, momentum progressively builds; conversely, when encountering resistance, momentum diminishes. Exploiting this conceptual framework is conducive to expediting convergence rates and ameliorating erratic fluctuations. In instances where the gradient direction remains constant, there is an accelerated speed of parameter updates. Conversely, when the direction of gradient undergoes alterations, the speed of parameter updates decelerates. The parameter update formula is as follows:

$$m_{t+1} = \beta \cdot m_t + \alpha \cdot \nabla J(\theta_t), \tag{6}$$

$$\theta_{t+1} = \theta_t - m_{t+1} \tag{7}$$

Specifically, β constitutes a momentum coefficient confined to the interval [0, 1), employed to govern the momentum's advancement rate. In addition, m_t denotes the momentum at the temporal point "t," while m_{t+1} signifies the momentum at the subsequent temporal point. It is evident from this formula that the parameter modification transpires under the joint influence of the current gradient and its previous counterpart. This interplay results in an amplification of momentum in the appropriate direction while facilitating its attenuation in the erroneous direction. Yurii Nesterov introduced the Nesterov Accelerated Gradient (NAG) method as an enhancement to the Momentum algorithm [11]. This particular refinement involves the substitution of the gradient of the loss function $\nabla J(\theta_t)$ with the gradient of the estimated position $\nabla J(\theta_t - \beta \cdot m_t)$ [9]. This alteration imbues the algorithm with a prospective aspect, thereby facilitating more precise parameter updates.

3) Adaptive learning rate algorithm Both the gradient descent method and the momentum approach utilize a constant learning rate, necessitating the ongoing refinement of this parameter to ascertain its optimal value. Nonetheless, distinct features or parameters frequently exhibit disparate dependencies on the learning rate, thus rendering the pursuit of an optimal learning rate a difficult and intricate task. As a response to this challenge, researchers have introduced algorithms capable of dynamically changing the learning rate to adapt variations in parameters. Examples of such algorithms encompass Adagrad [12], RMSProp, and Adam [13]. Adagrad is an extension of the gradient descent optimization algorithm, notable for its capacity to adaptively adjust the learning rate. This feature renders it particularly well-suited for the process of sparse data sets, as it facilitates larger updates for parameters that exhibit infrequent occurrences, while concurrently effecting more minor updates for parameters that manifest frequent appearances [9]. The mathematical formula of Adagrad is as follows:

$$s = s + \nabla J(\theta) \cdot \nabla J(\theta), \tag{8}$$

$$\theta = \theta - \frac{\alpha}{\sqrt{s + \epsilon}} \cdot \nabla J(\theta), \tag{9}$$

where s represents the accumulated sum of the squared gradient values, and ϵ denotes a minute constant introduced to avert division by zero in the denominator, typically on the order of 10e-8. Upon careful examination of the formula, it becomes evident that as the accumulated sum of the squared gradient experiences progressive increments throughout the training process, the corresponding learning rate undergoes a gradual reduction. This phenomenon has the potential to engender a noteworthy deceleration in the rate at which the model parameters are updated during the latter phases of training. Consequently, this deceleration could precipitate premature convergence of the training procedure, thereby impeding the model's capacity to effectively acquire optimal representations. To address the challenge associated with the swift decay of the learning rate, RMSProp has introduced enhancements. Specifically, RMSProp incorporates an additional parameter during the computation of the cumulative sum of gradient squares, thereby effecting a transformation into an exponentially weighted moving average of squared gradients, which is implemented as: $s = \lambda \cdot s + (1 - \lambda) \cdot \nabla J(\theta) \cdot \nabla J(\theta)$. The Adam algorithm amalgamates the principles of both Momentum and RMSProp techniques [13]. Adam algorithm integrates the estimation of the first-order moment and second-order moment of the gradient, thus optimizing the dynamic adjustment of the learning rate. This approach confers the benefit of employing an elevated learning rate during the initial phases of training, thereby expediting convergence. Subsequently, as training advances, the learning rate is gradually attenuated, serving to enhance and stabilize the convergence process. In contrast to the Momentum algorithm, Adam incorporates an adaptive learning rate. In comparison to RMSProp, it introduces momentum, thereby enhancing the stability of parameter updates.

In this project, the optimization methodology employed is Adam. The designated learning rate is established at 0.001, while the epsilon value is defined as 1e-8. Notably, the remaining parameters retain their default configurations.

2.2. Quantized Neural Networks (QNNs)

As the challenges faced by ANNs grow increasingly intricate and arduous, the architecture of neural networks correspondingly becomes more complex, necessitating a significant augmentation in the number of parameters employed. A noteworthy instance can be found in the emergence of the Chatbot GPT, which has recently garnered considerable acclaim for its remarkable achievements. This prodigious deep learning model builds upon the Transformer architecture, with GPT-3.5 boasting a staggering 175 billion parameters. However, the adoption of such large neural networks results in diminished operational speed and heightened storage demands, thereby presenting considerable obstacles to their deployment in mobile or embedded devices. Quantization of neural networks is one of efficacious approaches for addressing these challenges, garnering increasing attention in recent years. The essence of quantization lies in the conversion of neural network parameters from their native full-precision representation (float32) to low-precision representation, such as low-bit integers and fixed-point numbers.

This methodical transformation can greatly decrease memory usage and computation requirements, while increasing the processing speed.

Academic researchers have proposed many quantization technologies to respond different needs. Hubara et al. [14] introduced a method to train neural networks with extremely low precision weights and activation functions. Especially, training neural networks with only 1-bit weights and activations leads to Binarized Neural Networks (BNNs), which significantly improves the computing speed and reduces the memory consumption [14]. Micikevicius et al. [15] proposed a mixed precision training method which uses half-precision floating point numbers to train and maintains a copy of single-precision weights. The mixed precision method can be applied widely. For example, the method can be used in neural networks with more than 100 million parameters [15]. Han et al. [16] combined pruning, quantization and Huffman coding to compress the whole neural network models without reducing the accuracy, not merely replaced weights and operations with low precision.

In industry, many companies proposed their own quantization methods, such as TensorRT [17], IAO [18], and GLOW [19]. The basis of these quantization methods entails converting floating-point values, associated with inputs, outputs, weights and biases, into n-bit integers representations. Mathematically, the quantization and de-quantization can be defined as:

$$x_q = round(x_f/s + z), \tag{10}$$

$$x_f = s(x_q - z), \tag{11}$$

where x_q is a n-bit integer within $[m_q, n_q]$, x_f is a floating-point number within $[m_f, n_f]$, s is the scale factor, and z is the zero point. The scale factor is a positive real number employed in the transformation of a continuous range of floating-point values into a discrete range of integer values, maintaining the relative relationships among the values. The zero point refers to an integer that corresponds to the floating-point representation of zero within the aforementioned quantized domain. Therefore, it is imperative to ensure that the correspondence between m_f and m_q is established, as well as the correspondence between n_f and n_q. They can be defined though the following equations:

$$m_f = s(m_q - z) \tag{12}$$

$$n_f = s(n_q - z) \tag{13}$$

$$s = \frac{n_f - m_f}{n_q - m_q} \tag{14}$$

$$\begin{aligned} z &= round(\frac{0}{s} + z) \\ &= round(z) \\ &= round(\frac{n_f m_q - m_f n_q}{n_f - m_f}) \end{aligned} \tag{15}$$

Furthermore, x_q may fall outside the range $[m_q, n_q]$, necessitating the incorporation of the following clipping function to avert the occurrences:

$$\begin{aligned} x_q &= clip(round(\frac{x_f}{s} + z), m_q, n_q) \\ &= \begin{cases} m_q, & if\ x_q < m_q \\ round(\frac{x_f}{s} + z), & if\ m_q \leq x_q \leq n_q \\ n_q, & if\ x_q > n_q \end{cases} \end{aligned} \tag{16}$$

Due to the significant computational requirements associated with neural networks, the quantization of arithmetic operations is crucial to improving processing efficiency and reducing memory requirements. The subsequent equations delineate the quantization process of multiplication:

$$c_f = a_f b_f \tag{17}$$

$$s_c(c_q - z_c) = (s_a(a_q - z_a))(s_b(b_q - z_b)) \tag{18}$$

$$c_q = \frac{s_a s_b}{s_c}(a_q - z_a)(b_q - z_b) + z_c \tag{19}$$

The quantization techniques employed for other arithmetic operations, such as addition, subtraction, and division, are similar to those utilized for multiplication. Another noteworthy aspect pertains to the quantization of the activation function. Activations are commonly non-linear; however, only the ReLU can be efficiently represented using a conditional (if-then-else) form [20]. Employing direct quantization for other functions, such as Sigmoid and Tanh, may result in performance deterioration [21]. Therefore, other functions are transformed into look-up tables to circumvent such issue. The ReLU quantization method can be directly defined as:

$$y_f = ReLU(x_f) \tag{20}$$

$$\begin{aligned} s_y(y_q - z_y) &= ReLU(s_x(x_q - z_x)) \\ &= \begin{cases} 0, & if\ x_q < z_x \\ s_x(x_q - z_x), & if\ x_q \geq z_x \end{cases} \end{aligned} \tag{21}$$

$$y_q = \begin{cases} z_y, & if\ x_q < z_x \\ \frac{s_x}{s_y}(x_q - z_x) + z_y, & if\ x_q \geq z_x \end{cases} \tag{22}$$

The previously described technique effectively transforms floating-point numbers within the neural network into low-bit integers. However, its applicability for neural network verification is limited. This limitation arises from the verification process's typical requirement of assuming a series of inputs. The mentioned method necessitates the calculation of all variable values for each iteration, resulting in a substantial number of computations. Consequently, this situation presents a significant challenge in

terms of verification time. An additional quantization technique involves the conversion of floating-point numbers into fixed-point numbers. This technology has found applications in Digital Signal Processors (DSPs) and neural networks, yielding commendable outcomes [22, 23]. As per the IEEE 754 standard, the floating-point format is formally defined as:

$$V = (-1)^s \times 2^{(e-\mathrm{b})} \times f, \tag{23}$$

where 'V' means the floating-point numbers, 's' denotes the sign bit, 'e' pertains to the exponent part, 'b' represents the bias, and 'f' corresponds to the fractional part (also named mantissa)[24]. Specifically, it is observed that when the variable 's' is equal to zero, the value of 'V' is a positive number. Conversely, when 's' takes a value of one, 'V' assumes a negative magnitude. Furthermore, it is noteworthy that the function 'f' exhibits a range spanning from the inclusive lower bound of one to the exclusive upper bound of two, denoted as [1, 2). In relation to the data type float32, it is characterized by a sign bit occupying 1 bit, an exponent segment occupying 8 bits, a fractional segment occupying 23 bits, and the biases value set at 127 (computed as $2^7 - 1$). In contrast to floating-point numbers, fixed-point numbers exhibit a static fractional part instead of a dynamic one. Generally, fixed-point numbers are denoted by three components, namely the sign bit (S), the integer bit (I), and the fractional bit (F) [21]. The inter-conversion process between floating-point numbers and fixed-point numbers is defined as follows:

$$\text{fixed-point} = \text{floating-point} * 2^{\mathrm{F}}, \tag{24}$$

$$\text{floating-point} = \text{fixed-point}/2^{\mathrm{F}} \tag{25}$$

For example, the application of 8-bit fixed-point quantization is considered for the quantization of the numerical value 5.375. The 8-bit fixed-point representation is comprised of one sign bit, four integer bits, and three fractional bits. Consequently, the quantized result is 43 (5.73×2^3). Notably, upon subsequent dequantization, the outcome remains consistent with the original value, retaining its original form at 5.375 ($43/2^3$). The absence of error in the quantization of the number 5.375 can be attributed to the fact that its fractional part (0.375) perfectly aligns with the quantization precision (0.125). Nevertheless, numbers that fail to satisfy this condition will inevitably exhibit quantization errors. For instance, after undergoing the processes of quantization and dequantization for 1.231, the resultant value is 1.25, which shows the potential discrepancies introduced during the quantization process. The precision of a fixed-point number is contingent upon its fractional bits (F), while the range of values it can represent is associated with its integer bits (I). For instance, consider the case of an 8-bit fixed-point representation, where the most significant bit serves as the sign bit:

- When I=4 and F=3, the precision attains 0.125, and the range extends from -16 to 15.875.
- When I=5 and F=2, the precision achieves 0.25, and the range spans from -32 to 31.75.
- When I=6 and F=1, the precision equals 0.5, and the range encompasses -64 to 63.5.

Evidently, a greater number of fractional bits leads to heightened precision, while an increase in the number of integer bits results in a broader representable range. Hence, it is imperative to meticulously select the quantization bits in order to avert the occurrence of floating-point numbers surpassing the representable range of fixed-point arithmetic, and to ensure an adequate level of precision for the representation of floating-point numbers. This precautionary measure assumes utmost significance as it plays a pivotal role in mitigating potential information loss.

Quantization engenders a series of issues, and the most significant of which is the reduction in accuracy and the difficulty in verifying. Research findings have indicated that the application of quantization to neural networks leads to heightened challenges in the process of verification [25, 26]. Specifically, the verification of ANNs has been shown to be an NP-complete problem, whereas the verification of QNNs exhibits a higher level of complexity, falling under the category of PSPACE-hard problems [25, 26]. A straightforward illustration is that a variable which satisfied to the safety property exhibits a violation of said safety property after quantization. Figure 3 shows a simple neuron, where ReLU is the activation function and the bias sets to zero. The output of the neuron is $f(x_1, x_2) = ReLU(2x_1 + 3x_2)$. Assuming that $f(x_1, x_2) > 2$, and the output $f(0.147, 0.573) = 2.013$ satisfies the aforementioned assumption. However, after quantizing the neuron with 4 bits for the integer part and 4 bits for the fractional parts, the output alters to 1.3125 in decimal, which violates the assumption. Therefore, it is imperative to circumvent quantization errors while converting neural networks to low-precision format.

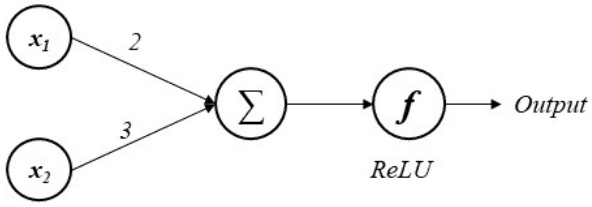

Figure 5: A simple neuron with ReLU as activation and without bias

At present, there are predominantly two quantization techniques: post-training quantization and quantization-aware training. Post-training quantization refers to preserve the neural network's weights and biases in a unchanging state after the completion of model training, then converting the numerical values of weights, biases, input, and output from high-precision representations to low-precision representations [27]. On the other hand, quantization-aware training pertains to the incorporation of quantization during the training process, enabling the neural network to learn and acquire proficiency in accommodating the information loss induced by quantization. [27]. Generally, post-training quantization will unavoidably introduce quantization errors; nonetheless, it offers the advantage of straightforward implementation and verification. Conversely, quantization-aware training exhibits superior accuracy and precision by virtue of its inherent characteristics, yet its verification poses challenges. Considering

the specific objectives of this project, post-training quantization is deemed a more suitable approach. Therefore, this research employed post-training quantization as a method to convert continuous floating-point representations into discrete fixed-point numbers.

2.3. Satisfiability Modulo Theories (SMT)

Prior to introducing SMT, it is necessary to review the concept of Boolean Satisfiability Problem (SAT). SAT is a fundamental problem in the field of computer science, which aims to determine whether there exists a set of value assignment for the variables in a propositional logic formula that would render the formula satisfiable [28]. For example, $a \wedge b$ is satisfiable as the formula is true when $a = TRUE$ and $b = TRUE$, and $a \wedge \neg a$ is unsatisfiable since the formula is always false.

SMT is an extension of SAT, whereby it combines SAT and first-order theories to provide a more expressive and powerful logic formula framework, that enables the modeling and solving of a wide range of problems [29]. Moreover, SMT is an efficient approach for verification. The subsequent code aims to interchange the values of the integer variables a and b, which can be verified through SMT.

```
int a0 = a;
int b0 = b;
int a1 = a + b;
int b1 = a1 - b;
int a2 = a1 - b1;
assert(a2 == b0 && b1 == a0)
```

The code can be converted into the following SMT logic formula:

$$\begin{aligned} &\forall a, b. a_0 = a \wedge b_0 = b \wedge a_1 = a + b \wedge \\ &b_1 = a_1 - b \wedge a_2 = a_1 - b_1 \\ &\Rightarrow \\ &a_2 = b_0 \wedge b_1 = a_0 \end{aligned} \tag{26}$$

Then, the formula can be eliminated by substitution method, resulting in:

$$\begin{aligned} &\forall a, b. a_2 = (a_0 + b_0) - ((a_0 + b_0) - b_0) \wedge \\ &b_1 = (a_0 + b_0) - b_0 \\ &\Rightarrow \\ &a_2 = b_0 \wedge b_1 = a_0 \end{aligned} \tag{27}$$

The formula can be finally simplified to:

$$\begin{aligned} &\forall a, b. a_2 = b_0 \wedge b_1 = a_0 \\ &\Rightarrow \\ &a_2 = b_0 \wedge b_1 = a_0 \end{aligned} \tag{28}$$

It is evident that the formula is valid, thereby confirming the verification of the code's capability to interchange the values of variables a and b. The aforementioned example illustrates the utilization of SMT for program verification, while for intricate tasks, it is recommended to employ SMT solvers, such as Z3 [30], cvc4 [31], Boolector [32], and Yices [33].

In this study, an series of SMT solvers are employed for the purpose of verification, with a specific focus on assessing and contrasting the differential impacts exerted by distinct SMT solvers on the verification time.

2.4. Safety properties for verifying ANNs and QNNs

Safety properties refer to the attributes and states of a system that guarantee secure functioning and prevent adverse consequences [34]. Within the realm of artificial intelligence, machine learning, and neural networks, safety properties aim to ensure the dependable, secure, and robust performance of these computational models. Specifically, during the verification process of neural networks, the safety properties are typically delineated separately for input and output, owing to the opaque nature inherent to neural networks [21]. Sena et al. [21] proposed a form that defined safety properties pertinent to the verification of ANNs and QNNs:

$$x \in \mathbb{X} \Rightarrow f(x) \in \mathbb{Y}, \tag{29}$$

which implies that if an input vector (x) resides within the designated input region ($\mathbb{X}$), the corresponding output ($f(x)$) is consequently situated within the specified output region ($\mathbb{Y}$). The definition of security properties consequently corresponds to the selection of input and output regions.

In particular, the input regions can be expressed using the subsequent formula:

$$\mathbb{X} = \{n : d_p(n, n_c) \leq c\}, \tag{30}$$

where n_c is a center point within the input domain, and $d_p(n - n_c)$ is the distance between the center point (n_c) and its surrounding neighbor points (n) [21]. The distance function is characterized by the L_p-norm, where typical values for p include 1, 2, and positive infinity. These norms are explicitly defined by the following equations:

$$d_p(n, n_c) = \sum_{i=1}^{j} |n - n_c|, \qquad where\ p = 1 \tag{31}$$

$$d_p(n, n_c) = \sqrt{\sum_{i=1}^{j} (n - n_c)^2}, \qquad where\ p = 2 \tag{32}$$

$$d_p(n, n_c) = max_i(|n^i - n_c^i|), \qquad where\ p = \infty \tag{33}$$

L_1-norm is the Manhattan distance, which represents the cumulative sum of the absolute values of the element-wise disparities between a pair of distinct points. L_2-norm is the Euclidean distance, which corresponds to the square root of the accumulated sum of the squared differences between corresponding elements of two distinct points. L_∞-norm is defined as the maximum value of the absolute differences between corresponding elements of two distinct points. Figure 4 depicts a graphical interpretation of the Lp norm within a two-dimension space, illustrating the cases for p equal to 1, 2, and infinity.

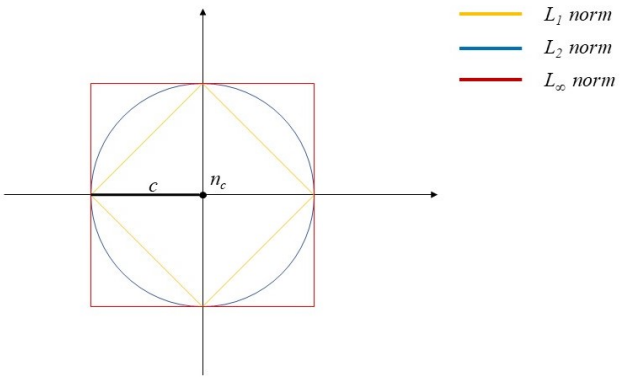

Figure 6: Graphical interpretation of L_p-norm

The definition of the output region is contingent upon the nature of the problem to be addressed. In general, problems can be categorized into two primary types: regression tasks and classification tasks. In regression tasks, the method for selecting the output region parallels that of the input region, defined as all points situated within a distance (c) from the output ($f(x)$) [21]. For classification tasks, the output region can be defined by considering the output values generated by the output layer, which signifies that the output value corresponding to the actual label surpasses all other output values [21]. In this study, safety properties of different sizes are utilized to verify and comprehensively assess their impact on both the temporal aspects of verification and the verification outcomes.

2.5. Existing SMT-based methods for verifying ANNs and QNNs

Currently, academic researchers have put forth an array of methodologies aimed at addressing the problems of verifying neural networks. These methodologies mainly encompass formal verification, the generation of adversarial samples, and verification grounded in SMT. This section predominantly elucidates the extant SMT-based approaches pertaining to the verification of ANNs and QNNs.

Katz et al. introduced an algorithm with a primary focus on the verification of neural networks employing the Rectified Linear Unit (ReLU) as the activation function [35]. The algorithm, denoted as "Reluplex," demonstrates a methodology of employing the inherent piece-wise linear characteristics of the ReLU function to model the verification of neural networks into linear programming (LP) or mixed integer linear programming (MILP) [35]. Furthermore, Katz et al. devised an automated reasoning framework known as "Marabou," specifically engineered for verification and analysis of neural networks [36]. This tool adeptly translates network property inquiries into constraint satisfaction problems, thereby facilitating the comprehensive verification of neural networks encompassing diverse activation functions. In contrast to the verification processes applied to artificial neural networks, the verification procedures pertinent to quantized neural networks tend to present heightened intricacy. Sena et al. opted for the utilization of fixed-point numbers as opposed to their floating-point counterparts in the quantization of neural networks [21]. Subsequently, they introduced an approach for the quantification of nonlinear activation functions. This facilitates the effective verification of neural networks, encompassing activation functions beyond ReLU [21]. For a special type of quantized neural networks, specifically Binarized Neural Networks (BNNs), Narodytska et al. elucidate that the architecture of this neural network variant can be formalized into a Boolean formula representation [37]. This formulation implies that the verification of BNNs can be addressed through employment of SAT solvers [37]. This study employs fixed-point numbers for the quantification of neural networks, then utilizes model checking tools based on SMT to verify ANNs and QNNs.

3. Research Methodology

This segment delineates the implementation methodology employed in this project, comprising three primary components. The initial segment involves the training of the neural network followed by its transformation into a C language program. Subsequently, the second segment encompasses the bidirectional conversion mechanisms between floating-point and fixed-point numerical representations, alongside the formal exposition of fixed-point arithmetic principles and the neural network quantization process. Lastly, the third segment pertains to the precise delineation of safety properties, accompanied by the subsequent verification process involving the quantized neural network through employment of ESBMC.

3.1. ANN implementation

The implementation of ANNs can be divided into two primary phases. The initial phase involves utilizing the Python programming language to conduct the training and evaluating of the neural network, subsequently saving both the resultant model and its associated parameters. The subsequent phase encompasses the transformation of the neural network model, post-training, into a model scripted in the C programming language.

Currently, a multitude of deep learning frameworks for the Python programming language are available, designed to facilitate the realization of neural network implementations, such as TensorFlow, PyTorch, and Keras. This project employed the Keras framework to construct ANNs tailored for the purpose of executing classification tasks. Keras is a neural network Application Programming Interface (API) integrated within the TensorFlow platform, which offers many functionalities catering to training, experimentation, and analytical pursuits. Developing a high-performance ANN entails meticulous optimization of its architecture and the fine-tuning of associated parameters. The architecture of a neural network can typically be categorized into three primary components: the input layer, hidden layer(s), and output layer. The input and output layers are designed to align with the features of dataset used and the specific objectives of the task being addressed. In the context of the MNIST multi-class classification task, the dataset consists of images with dimensions of 28∗28 pixels and encompasses ten distinct categories. Consequently, the input layer comprises 784 neurons, which corresponds to the product of the image dimensions (28∗28), and the output layer contains 10 neurons, with each neuron representing one of the ten distinct categories. Furthermore, the size and number of hidden layers are contingent upon the complexity of problem. While no determined solution exists, several general guidelines can be employed to delineate the structure of hidden layers. The size and number of hidden layers should be appropriately chosen, as excessively large or small configurations may result in overfitting or underfitting, respectively, adversely affecting the model's generalization capabilities. A neural network comprising fewer than four hidden layers is typically adequate for addressing the majority of tasks [38]. Nevertheless, it is essential for the network to have at least one hidden layer, as ANNs devoid of hidden layers are constrained to represent linearly separable functions or decisions. The number of neurons in the hidden layer is generally determined by taking into account the sizes of both the input and output layers. There are some common strategies, such as $H = S/a(N_i + N_o)$, $H = \sqrt{N_i + N_o} + a$, $H = \sqrt{N_i * N_m}$, and $H = \log_2 N_i$, where H is the number of hidden neurons, a is a constant within $[1, 10]$, S is the amount of data, N_i is the number of input neurons, and N_o is the number of output neurons. H can be initially ascertained using the aforementioned strategies, followed by a progressive adjustment to enhance the network's performance. However, it is important to emphasize that, for this project, maintaining a minimal level of complexity in hidden layers while still achieving satisfactory performance is crucial. This is due to the fact that neurons within the hidden layer

employ nonlinear activation functions to discern intricate patterns, and the quantity of non-linearities constitutes the primary determinant of verification time [21]. Furthermore, the judicious selection of appropriate hyperparameters holds significant import within the realm of neural networks. Paramount among these hyperparameters in the present study are the batch size and epochs. Batch size denotes the requisite volume of samples for a singular training iteration. While a larger batch size has the potential to expedite the training velocity, it carries the inherent risk of diminishing the model's capacity for generalization. Conversely, opting for smaller batch sizes may engender enhanced outcomes but could protract the convergence period. The Epochs parameter delineates the number of complete passes the learning algorithm undertakes across the entire training dataset. Throughout the iterative progression of the gradient descent model's training, the neural network transits gradually from an underfitting state to an optimal fitting state, subsequently culminating in an overfitting state post-attainment of optimality. Hence, a definitive choice for the ideal epoch value remains elusive, and it needs to be adjusted according to the situation. Excessive epoch values might precipitate overfitting, whereas excessively epoch values might yield underfitting manifestations. Upon establishing the network architecture and hyperparameters, it is essential to proceed with the training and evaluation process, which involves fine-tuning pertinent parameters, including weights and biases, to optimize the model's performance.

Prior to commencing neural network training, it is imperative to undertake data preprocessing procedures. Specifically, in the case of grayscale images, data normalization is employed to transform the eigenvalues of the samples into a confined range of [0, 1]. This procedural measure yields several advantageous outcomes, including the augmentation of training expediency, enhancement of model robust, and facilitation of expedited convergence towards an improved local optimum. As an illustration, for the MNIST dataset, the normalization procedure can be instantiated through the utilization of the code: `MNIST.astype('float32') / 255.0`. Subsequently, the dataset is partitioned into a training set and a validation set utilizing the 'train_test_split' function inherent in the 'sklearn' library. As an instance, to allocate 20% of the dataset for validation and 80% for training, the ensuing code suffices: `train_test_split(data, label, test_size=0.2)`. During the training of the neural network, the project invokes the ModelCheckpoint function found within the keras.callbacks module. This function facilitates the preservation of distinct stages of the model throughout the training procedure. In this project, it enables to save the model exhibiting the best accuracy rate concerning the validation set throughout said training phase. The following code exemplifies its implementation: `ModelCheckpoint(file_path, monitor='val_accuracy', verbose=1, save_best_only=True)`. Upon the accomplish of neural network training, the subsequent step involves utilizing the 'load_model' function within the 'keras.models' framework to ingest the preserved model. This is followed by the utilization of the 'get_weights' function to retrieve the parameters encompassing weights and biases embedded within the network architecture. This function is designed to return a list, wherein each constituent array corresponds to the weights or

biases pertaining to a specific layer and its contiguous layer. To illustrate, in the scenario of a neural network featuring a singular hidden layer, the resultant list engendered by this function encompasses four elements. The first element pertains to the weights extending from the input layer to the concealed layer, the second element encapsulates the biases of the hidden layer, the third element entails the weights originating from the concealed layer and extending to the output layer, the final element represents the biases of the output layer.

The subsequent step involves the conversion of the trained models into the C language. It is noteworthy that Python operates as an interpreted language, wherein the Python interpreter undertakes a sequential line-by-line interpretation of the code during running time. Conversely, the C language is categorized as a compiled language, wherein the C compiler directly translates the provided C source code into machine-readable code. Notably recognized for its commendable execution efficiency, the C language particularly excels in tasks demanding significant computational prowess. Given that the verification of neural networks frequently necessitates extensive computational operations, the employment of the C language is more advantageous for such verification endeavors as compared to the Python language. Additionally, the realm of neural network verification is substantially facilitated by a multitude of verification tools, including SMT solvers and supplementary formal method tools. Many of these tools are equipped with interfaces or APIs that are predominantly designed within the ambit of the C language. These provisions significantly expedite the intricate process of verifying neural networks, underscoring the pertinence of utilizing the C language in this domain. This study used two method to transmute the neural network model into the C language, encompassing both automatic and manual conversion methodologies.

In the context of automatic conversion, the process integrates the utilization of the Keras2C library, which facilitates the transformation of neural network models fabricated within the Keras framework into executable code scripted in the C programming language [39]. This Python-based library supports various Keras layers, such as Dense, Dropout, and BatchNormalization, enabling the transformation of numerous neural network architectures. Keras2C offers the utility of the 'k2c' function, designed for the transformation of the forward propagation segment of a Keras model into a corresponding C function [39]. This conversion process is executed through the utilization of the following code: `k2c(saved_model, c_function)`. In particular, the resulting tensor within the converted C function is instantiated employing the 'k2c_tensor' data type [39]. This data type finds its definition within Figure 7. This structure comprises four distinct variables: 'array,' 'ndim,' 'numel,' and 'shape.' The variable 'array' serves as a pointer directed towards an array consisting of floating-point numbers, wherein it retains the flattened tensor values in a row-major order [39]. 'ndim' is denoted as an unsigned integer, functioning to signify dimensional attributes of the tensor. Similarly, 'numel,' also an unsigned integer, functions as a representation of the total count of elements present within the tensor. Lastly, the variable 'shape' assumes the form of an array of integers, effectively encapsulating the magnitude

```
1  struct k2c_tensor
2  {
3      float *array;
4
5      size_t ndim;
6
7      size_t numel;
8
9      size_t shape[K2C_MAX_NDIM];
10 };
```

Figure 7: Definition of the datatype 'k2c_tensor'

of the tensor along each of its respective dimensions.

For diminutive neural networks, manual conversion is more appropriate, as automated conversion utilities might engender code redundancies to a relatively greater extent. Manual conversion enables the tailoring of the model's transformation, affording precision in code generation, alignment with particular requisites, and mitigation of superfluous resource expenditure. To effectuate manual conversion, the initial phase involves the utilization of the C programming language to construct an analogous framework to that of the neural network. Subsequently, the ensuing phase encompasses the utilization of the learned neural network model's weight coefficients and bias terms to instantiate the pertinent parameters. By way of illustration, in the case of a neural network with structure 3x5x3, its corresponding C language implementation is exemplified in Figure 8. In the presented code excerpt, two custom functions, denoted as 'neuron' and 'find_max_index', are employed. The precise manner in which these functions are implemented is expounded in Figure 9. The primary role of the 'neuron' function is the computation of neuron output. Precisely, this function accommodates four parameters: firstly, the weight originating from the antecedent neuron layer to the present neuron; secondly, the input; thirdly, the dimensions of this input; and finally, the inherent bias of the neuron. Subsequently, a multiplication operation is performed, associating each weight with its corresponding input. The cumulative summation of these products yields the unactivated output. This resultant is subsequently subjected to an activation function, culminating in the actual output. The 'find_max_index' function is employed within the context of the output layer to compute the predicted value. This function is parameterized by two variables: the output value from the output layer i.e. 'arr', and the dimension of said output layer i.e. 'size'. The fundamental operation of this function involves the computation of the index associated with the maximum value present within the output layer. Subsequently, this computed index is returned, thereby serving as the predicted value.

```
int main(){
    float input[3];

    float layer1[4];

    float weight_1_0[3] = {-0.25f, 0.02f, 0.11f};
    float bias_1_0 = 0.0f;
    layer1[0] = neuron(weight_1_0,input,3,bias_1_0);

    ...

    float layer2[3];

    float weight_2_0[4] = {-0.24f,0.41f,0.65f,0.08f};
    float bias_2_0 = 0.16f;
    layer2[0] = neuron(weight_2_0,layer1,4,bias_2_0);

    ...

    int predict_class = find_max_index(layer2, 3);
}
```

Figure 8: C language implementation for $3 \times 4 \times 3$ neural network

3.2. Quantization for neural networks

The quantization process of the neural network encompasses a tripartite progression, involving the translation between floating-point and fixed-point numerical representations, the quantization of mathematical operations, and the incorporation of quantization principles within the neural network model.

Initially, the data type ascribed to fixed-point numbers is int64, with the corresponding implementation denoted as follows: `typedef int64_t fxp`. As delineated in section 2.2, the process of quantifying floating-point numbers into fixed-point numbers, possessing n decimal bits, necessitates the multiplication of the floating-point values by the nth exponent of 2. Subsequently, a type conversion is executed to effectuate the transformation into the fxp domain. The specific code implementation is articulated as: `(fxp)(x * pow(2, n))`. Conversely, for the dequantization procedure, the requisite involves dividing the fixed-point numbers by 2 raised to the power of n, culminating in a type conversion to the float32 category. The precise code manifestation is expressed as: `(float)(y / pow(2, n))`. However, both quantization and dequantization processes entail exponential operations, which exhibit

```
float neuron(float *w,
             float *input,
             int size,
             float b)
{
    float output = 0.0f;

    for (int i = 0;
         i < size;
         ++i) {
        output =
            output+w[i]*input[i];
    }

    output = output + b;

    output = activation(output);

    return output;
}
```

(a) Neuron Function

```
int find_max_index(float *arr,
                   int size)
{
    int max_index = 0;

    for (int i = 1;
         i < size;
         ++i) {
        if (arr[i]>arr[max_index])
        {
            max_index = i;
        }
    }

    return max_index;



}
```

(b) find_max_index Function

Figure 9: Custom functions

inefficiency and impose significant memory demands. The operational sequence of the neural network necessitates recurrent quantization and dequantization, thereby exacerbating concerns regarding memory consumption and temporal inefficiency. Hence, the utilization of a look-up table is used to this project, serving as a substitution for the exponential operation. This look-up table comprises two arrays, each containing 32 elements, employed for the processes of quantization and dequantization. The quantization array stored the scale factor, with each element being representative of the nth exponentiation of 2, where 'n' denotes the corresponding index of said element. The corresponding code is denoted as follows: `double q_scale[31] = {1, 2, 4 ...}`. The dequantized array is similar to the aforementioned structure, differing only in that each constituent element transforms into the reciprocal of 2 raised to the power of 'n,' again with 'n' signifying the associated index of the element. The corresponding code is expressed as: double `d_scale[31] = {1, 0.5, 0.25 ...}`. Once the lookup table has been established, the quantization and dequantization codes undergo modification to incorporate the respective functions with the scaling factor, `(fxp)(x * q_scale[n])` and `(float)(y * d_scale[n])`. Furthermore, it is imperative to guarantee that the floating-point representation remains

within the bounds of the fixed-point numerical range delineated by the predetermined number of integer bits. Consequently, an imperative task involves the computation of both the upper and lower bounds of neuronal output within the neural network, subsequently ascertaining the minimum integer precision required to sustain this range. Nonetheless, it is plausible that overflow might ensue during the computation of forward propagation, thereby necessitating the establishment of a dedicated function tailored to address such occurrences. The check_overflow function is defined in Figure 10. This function

```
1 fxp check_overflow(fxp x){
2     if (x < fxp_min){
3         return fxp_min;
4     }
5     else if (x > fxp_max){
6         return fxp_max;
7     }
8     return fxp(x);
9 }
```

Figure 10: The "check_overflow" function

bears resemblance to the clip function found within the numpy library, wherein values are constrained within a predefined range. In the context of fixed-point numbers surpassing the upper bound, they are equated to the maximal representable value for said fixed-point numerical format. Similarly, for fixed-point numbers residing beneath the lower bound, they are equated to the smallest representable value for the given fixed-point numeric representation. In instances where the most significant bit assumes the role of a sign bit, denoted as 'S', the integral part is denoted as 'I', and the fractional part is denoted as 'F', the minimum value attainable within this fixed-point system, `fxp_min`, is calculated as -2^{I+F}, while the maximum value, `fxp_max`, corresponds to $2^{I+F} - 1$.

The subsequent phase entails the quantification of arithmetic operations, primarily encompassing addition and multiplication. This pertains to the fact that during the forward propagation of the artificial neural network, computations are restricted to the determination of the product between weights and their corresponding inputs, followed by the summation of products. The addition of fixed-point numbers is similar to the process employed for the addition of floating-point numbers. Notably, a discerning distinction lies in the imperative to meticulously ascertain potential instances of result overflow. This distinction is conspicuously manifested in Figure 11, wherein the presented example illustrates the implementation of fixed-point numerical addition within the C programming language. The process of multiplying fixed-point numbers markedly diverges from the multiplication of floating-point numbers. In the context of fixed-point multiplication, a direct multiplication of two fixed-point numbers is insufficient. In addition, meticulous consideration must be given to the manipulation of the fractional

```
1 fxp add(fxp a, fxp b){
2     fxp result;
3     result = fxp((fxp)(a) + (fxp)(b));
4     result = check_overflow(result);
5     return result;
6 }
```

Figure 11: Fixed-point numbers addition

component, involving rounding operations to appropriately align the decimal point in accordance with a predetermined precision. This intricacy of fixed-point multiplication is elucidated in Figure 12, which shows the process of fixed-point multiplication. The sixth line of code presented in Figure 12 delineates

```
1  fxp multiply(fxp a, fxp b){
2      fxp result, fresult;
3      result = fxp((fxp)(a) * (fxp)(b));
4      if (result >= 0){
5          fresult =
6              (result +
7              ((result & 1 << (f - 1)) << 1)) >> f;
8      }
9      else{
10         fresult =
11             -(((-result) +
12             (((-result) & 1 << (f - 1)) << 1)) >> f);
13     }
14
15     fresult = check_overflow(fresult);
16     return fresult;
17 }
```

Figure 12: Fixed-point numbers multiplication

the procedural steps associated with the rounding of positive fixed-point numbers, whereas the tenth and eleventh lines elucidate the analogous rounding procedure applied to negative fixed-point numbers. Specifically, the expression `(1 « (f - 1))` computes a numerical value wherein the most significant bit is set to 1, while all other bits are set to 0. This value is subsequently subjected to a bitwise AND operation with the fractional component of the result. The objective of this operation is to determine whether the fractional component of the result extends beyond the anticipated bits. Should the most

significant bit be found to be 1, it signifies an overage in the decimal fraction, necessitating an offset (achieved through a left shift) to be appended to the excessive decimal fraction. Subsequent steps involve rounding procedures and right shifts, orchestrated to reposition the fractional part to its initial position. The culmination of these processes engenders an approximation of a rounded fixed-point number.

The final stage of the quantization process involves the integration of the implemented quantization operations into the neural network architecture, accompanied by requisite refinements to both the structural and functional aspects of the neural network. Initially, it is imperative to effectuate a modification in the data type of the array employed for the retention of the output from every layer of neurons, transitioning it from a floating-point configuration to a fixed-point configuration. Subsequent to this alteration, requisite adaptations shall be requisite for associated bespoke functions, inclusive of but not confined to the "neuron" and "find_max_index" functions expounded upon within the confines of section 3.1. Figure 13 depicts the adapted "neuron" function, wherein a distinct alteration is implemented. Specifically, the function's return value type is transmuted from "float" to "fxp." The

```
fxp neuron(float *w,
           float *input,
           int size,
           float b)
{
    fxp output = 0;

    for (int i = 0; i < size; ++i) {
        fxp w_q = float_to_fxp(w[i]);
        fxp i_q = float_to_fxp(input[i]);
        output = add(output, multiply(w_q, i_q));
    }

    fxp_t b_q = float_to_fxp(b);

    output = q_activation(add(output, b_q));

    return output;
}
```

Figure 13: The "neuron" function used in QNNs

data type of each float parameter transmitted into this function undergoes a conversion to "fxp". Furthermore, the antecedent arithmetic computations have been supplanted with fixed-point arithmetic operations. Concurrently, the activation procedure employs quantized activation functions. Nevertheless,

the applicability of this function is confined to the computation of neuronal outputs within the first hidden layer. This limitation stems from the fact that neurons situated in the first hidden layer utilize the input layer as their primary input source, with said input data exhibiting a floating-point data type. In contrast, the subsequent hidden layers and the output layer are characterized by an input data type which is fxp. As a result, it becomes imperative to develop a new function to ensure the valid computation of neuronal outputs in the subsequent layers. The new function closely resembles its predecessor, diverging solely in the alteration of the data type of the parameter "input". Moreover, the section of code responsible for the conversion of input from float to fxp (observable in the 10th line of Figure 13) is rendered redundant and thus, expunged. Furthermore, the requisite modification for the "find_max_index" function pertains to the incorporation of additional code aimed at effecting the dequantization process on the data yielded by the output layer, thereby facilitating its conversion into a floating-point representation. An additional aspect meriting attention pertains to the quantization of the activation function. In the case of the ReLU function, quantization refers the conversion of zeros within floating point representations into their fixed point counterparts. In contrast, functions such as the Sigmoid and Tanh, which necessitate exponential computations, are discretized into lookup tables to circumvent intricate exponential operations, thereby augmenting computational efficiency. The fundamental principle of the approach involves the a prior computation of function values at a series of discrete points. Subsequently, the nearest discrete point relative to the input value is determined, thereby resulting an approximate function value.

3.3. QNN verification

In this project, the verification of quantized neural networks is bifurcated into dual constituent segments. The initial segment encompasses the definition of the safety property predicated upon the dataset, followed by the subsequent annotation of said safety property onto the C language model. The second segment involves the utilization of the Efficient SMT-Based Bounded Model Checker (ESBMC) for the verification of the aforementioned annotated model.

The safety property is divided primarily into input and output regions. Given the classification tasks focus of this experiment, the specification of the output region is relative simplicity. For instance, in cases where the actual label value of a given dataset is 0, the output region corresponds to both the predicted neural network value and the actual value. This is programmatically articulated as follows: `assert(predicted_value == 0)`. However, the definition of the input region is more intricate. It necessitates the identification of center points for each class of data, coupled with the establishment of upper and lower bounds for each feature within said class. When dealing with classification based on characteristic values, the selection of these center points can be accomplished through the utilization of a clustering algorithm. In this particular project, the chosen clustering algorithm is KMeans, an unsupervised learning method employed for the aggregation of data into K clusters. The algorithm

involves the initial selection of K points as centroids, followed by the computation of the distance between each data point in the dataset and these centroids. Subsequently, data points are assigned to the closest cluster based on the distance. A re-calibration of cluster centroids is performed by calculating the mean values of the data points within each cluster, which are then adopted as the new centroid coordinates. This iterative process, encompassing the aforesaid steps, persists until the algorithm converges upon an optimal solution. After the determination of the center point, the input domain can be delineated in accordance with the upper and lower bounds corresponding to the feature's value. As expounded upon in section 2.4, this input domain denotes the spatial expanse wherein the distance from the central point is constrained by a parameter denoted as "c." In the context of the present classification tasks, the parameter "c" conveys the proportion of the maximal span of inputs [21]. For example, consider the central point denoted as x_c = (0, 0), the utmost admissible range for x_1 being [-1, 1], and correspondingly, the maximum range allowable for x_2 being [-2, 2]. In a scenario wherein c equals 50 percent, the demarcated input domain manifests as follows: $-0.5 \leq x_1 \leq 0.5$, and $-1 \leq x_2 \leq 1$. For tasks pertaining to gray-scale image classification, the center points reside within the unaltered base images. This entails the absence of any modifications, encompassing elements such as extraneous noise, positional deviation, or rotational adjustment. The input domain comprises an assemblage of gray-scale images, derived through manipulation of pixel intensities within the foundational image. This manipulation involves increase or decrease adjustments to the grayscale values of each pixel, confined within a parameterized range denoted as 'c'. To illustrate, considering a scenario wherein the base image encompasses three distinct pixels, each characterized by gray-scale values of 0, 255, and 0. If 'c' is prescribed as 20, the resulting input domain would be defined as follows: [0, 40], [215, 255], [0, 40]. Moreover, it is imperative to utilize ESBMC-specific annotations for the precise specification of safety properties, thereby facilitating the valid verification of programs by ESBMC. The instruction `__ESBMC_assume(x)` assumes a pivotal role in delineating the input region, while the instruction `__ESBMC_assert(y, "text")` holds significance in demarcating the output region. In the latter instruction, the accompanying textual descriptor "text" encapsulates the informational content emitted by the program in instances where "y" fails to satisfy the prescribed conditions.

Subsequently, employ the ESBMC tool to conduct verification on the annotated C model. ESBMC is a model checker that specializes in the verification of C/C++ programs for all platforms [40]. This tool facilitates users in formulating specific program properties, subsequently enabling the assessment of the program's adherence to these properties [40]. Noteworthy is the tool's provision of an assortment of SMT solvers, namely Z3, Boolector, cvc4, and Yices, which are harnessed to encode the program in the form of an SMT formula. It is pertinent to acknowledge that ESBMC endows users with the flexibility to tailor its parameters to suit their particular requirements, thereby engendering optimization avenues for the verification procedure. The command for effecting the process of verification entails the utilization of the ESBMC tool in the following manner: `esbmc test_file.c -parameter_option`. Herein,

"test_file.c" embodies the C language model necessitating verification, while "parameter_option" signifies the designated parameter to be enabled, and multiple parameters can be enabled at the same time. Subsequent to the verification procedure, ESBMC will output the verification outcome, accompanied by the expended duration of the verification process.

4. Evaluation

This section provides an introduction to the dataset and experimental environment employed in this study. Furthermore, it assesses and discusses upon the influence of ESBMC parameters on the verification process, as well as the ramifications of quantification on the verification time and outcomes.

4.1. Benchmarks

This study employed two datasets for experimental investigation: specifically, the Iris dataset and the Chinese digit dataset. Within this subsection, primary focus is directed towards presenting comprehensive insights into both datasets, alongside their respective neural network architectures, safety properties, and the minimal bit-width employed for quantization purposes. Furthermore, the experimental setting in which these investigations were conducted is also expounded upon within this subsection.

4.1.1. Iris dataset. The Iris dataset is a classic collection of data for the purpose of conducting classification-based experimental test. This dataset is characterized by its inclusion of three distinct Iris classifications: Setosa, Versicolor, and Virginica. Each category encompasses precisely 50 individual samples, with each individual sample being composed of a quartet of distinct features concomitant with a corresponding label. These features encompass the dimensional values of sepals and petals, inclusive of both length and width, measured in centimeters. The label assigned to each sample corresponds to the respective Iris classification to which it pertains. The neural network architecture deployed for this dataset is $4 \times 10 \times 3$. Within this model, the ReLU is adopted as the designated activation function in hidden layers, while the output layer's activation function is stipulated as the softmax function. In addition, the categorical cross-entropy function is employed as the loss function, and the optimization of network weights is realized through the Adam optimization algorithm. Furthermore, the batch size parameter is defined at 32, delineating the number of samples utilized per iteration during training. Following an iterative process encompassing 37 epochs, the neural network demonstrates a notable achievement by attaining a validation accuracy of 100%. This achievement signifies the convergence of the network, indicative of its capability to effectively predict and classify Iris types with a remarkable degree of accuracy. The neural network's output range is bounded, with an upper bound of 10.4 and a lower bound of -6.3. As a consequence, the minimum integer bit required for quantization is determined

to be 4 bits. As described in section 3.3, the input regional range within the confines of the safety property is controlled by the parameter 'c'. For this dataset, 'c' represents a proportional measure relative to the maximal range of attainable feature values. In the course of this experimentation, the parameter 'c' assumes three values, specifically: 5, 10, 20.

4.1.2. Chinese digit dataset. The second dataset used is denoted as the "Chinese Digit Dataset." This dataset comprises grayscale images with pixel dimensions of 5 by 5, having been generated through the utilization of the Python programming language. Within this dataset, one can discern three distinct categories of Chinese digits, specifically designated as Chinese Character one, Chinese Character two, and Chinese Character three. It is noteworthy that each category of digit encompasses a total of 50 individual samples. The foundational image serving as the archetype is visually expounded upon in Figure 14. It is imperative to elucidate that the dataset is generated through multifarious

Figure 14: The base images of Chinese digit

processes, including the introduction of noise to the foundational image, as well as the implementation of translational and rotational transformations. In contrast to the Iris dataset, the present dataset employs a more complex network architecture denoted as $25 \times 12 \times 5 \times 3$. The choices pertaining to activation functions, optimizers, and related parameters remain consistent with those applied in the Iris neural network model. The range of output of neurons is confined within limits, characterized by an upper bound of 12.7 and a lower bound of -7.1. As a result, it becomes evident that the minimal integer bit for effective quantization is 4 bits. Furthermore, the parameter 'c' signifies the alteration in gray-scale values of the pixel. The magnitudes employed in this experimental context encompass 5, 10, 15, and 20.

4.1.3. Experimental environment. The primary experimental environment is shown as follows:

- CPU: Intel Core i7-12700KF @ 3.6GHz
- RAM: 32GB
- Operating System: Ubuntu 20.04.4
- Library: Numpy 1.24.3, Matplotlib 3.7.1, TensorFlow 2.9.0, Keras 2.9.0
- Tool: ESBMC v7.2

4.2. Evaluation on ESBMC parameters

ESBMC offers a range of parameters geared towards enhancing the efficiency of the verification process. This study focuses on the evaluation of three distinct categories of parameters: SMT solvers, property checking, and miscellaneous configurations. The criterion for evaluation is the comprehensive time of the verification process across both datasets. The quantification approach encompasses three levels: 4-bit, 8-bit, and 16-bit quantification. Each verification iteration is conducted thrice, and the resultant average is adopted as the representative verification time, thus mitigating potential errors stemming from extraneous variables. A reduction in verification time signifies the efficacy of the parameters in optimizing the verification process, whereas a nearly consistent or heightened verification time indicates a lack of substantial optimization influence by the parameters.

Regarding the choice of SMT solver, the verification procedure employs Z3, Boolector, Bitwuzla, and Yices in succession. While Z3 successfully verifies 4-bit quantized neural networks, it fails to complete the verification process when applied to 8-bit and 16-bit quantized neural networks. Consequently, Z3 is deprecated for this study. Conversely, the remaining trio of solvers effectively handle verification across all quantization scenarios, albeit with minor variations in verification times. Boolector manifests a verification time of 57.749 seconds, while Bitwuzla completes the process in 55.653 seconds. Notably, Yices demonstrates the shortest verification time which is 45.976 seconds. The performance of Boolector and Bitwuzla is nearly equivalent, with Bitwuzla exhibiting a slight superiority of 3.63%. Meanwhile, Yices surpasses both Boolector and Bitwuzla, boasting an enhancement of 20.387% over Boolector and 17.388% over Bitwuzla. Consequently, Yices is elected as the designated SMT solver for ensuing experimental endeavors.

For the "property checking" and "miscellaneous options" parameters, it is advised to activate and deactivate specific settings. To elaborate, the toggling of "no-bounds-check," "no-pointer-check," "no-div-by-zero-check," and "interval-analysis" warrants consideration. Upon activation of the "no-bounds-check" parameter, the array bounds are exempt from scrutiny, resulting in a verification time of 43.295 seconds. Furthermore, an ensuing boost of 5.83% in performance is discernible post-activation, thus warranting its perpetuation in subsequent iterations. Moreover, when engaging the "no-pointer-check" parameter, the verification procedure omits checking of pointer relationships, resulting in a verification duration of 50.841 seconds. However, no appreciable performance enhancement is manifest, thus rendering the activation of this parameter unnecessary. Analogously, activation of the 'no-div-by-zero-check' property circumvents scrutiny of divisor-nullity in divisions, culminating in a verification time of 43.176 seconds, with insignificant performance improvement. Consequently, the rationale for activating this parameter remains unsubstantiated. The activation of "interval-analysis" introduces additional assumptions conducive to delimiting the value range of internal variables during the verification process. Post-activation verification time records 42.363 seconds, accompanied by a minor augmentation in

performance, thereby validating the rationale for its activation. To recapitulate, in subsequent experimental endeavors, the parameters to be enabled encompass "no-bounds-check" and "interval-analysis," alongside the adoption of Yices as the designated SMT solver. The corresponding command takes the form: `esbmc test_file.c -no-bounds-check -interval-analysis -yices`.

4.3. Impact of varying quantization levels on verification

This study employs varying bit quantization schemes, ranging from 4 bits to 16 bits, to quantize the neural network. Subsequently, a comparative analysis is conducted on the verification time of these quantized neural networks across different bit-widths. For every bit schema, the verification procedure is iterated thrice for each class, and the resultant average is adopted as the representative outcome for that specific class. This aforementioned methodology is systematically executed for each verification category. The verification time for all classes within a given bit-width is meticulously recorded, enabling the subsequent computation of the average verification time corresponding to that particular bit-width. This calculated mean is then regarded as the result for the corresponding bit schema. Figure 15 depicts the verification time of three distinct safety properties across varying bit quantization

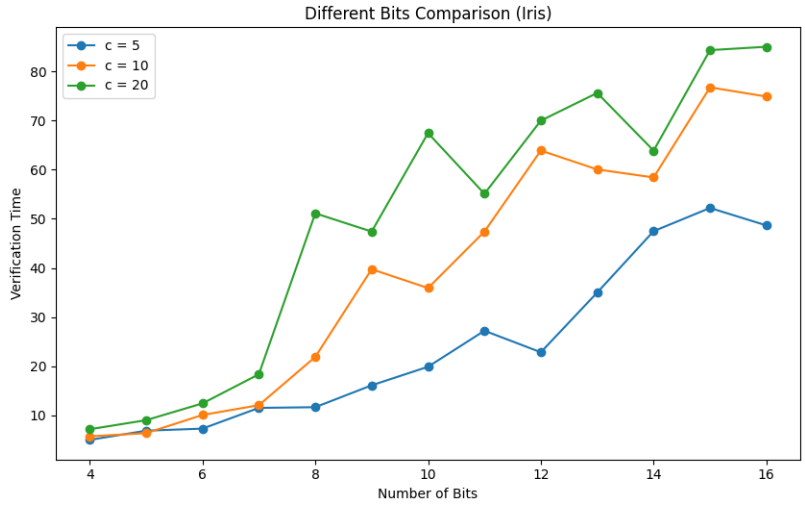

Figure 15: Verification time under different bits

configurations for the Iris dataset. From the experimental findings, it becomes apparent that an escalation in bits corresponds to a general increment in verification time. Nonetheless, this temporal increment is characterized by non-linear marked by oscillatory patterns. In the case where c=5, a progressive rise in verification time is discernible from 4 bits to 11 bits, leading from 4.989 seconds to 27.209 seconds. Similarly, within the range of 11 bits to 16 bits, the verification time exhibits increased oscillations, culminating in the verification time of 48.656 seconds. For c=10, the verification time experiences a gradual elevation from 4 bits to 9 bits, spanning from 5.707 seconds to 39.734 seconds. Moreover,

spanning the interval from 9 bits to 16 bits, the verification time demonstrates intensified oscillatory increment, characterized by an augmented oscillation amplitude, culminating in the verification time of 74.892 seconds. In the instance of c=20, the amplitude of oscillation in verification time further escalates, improving from 7.154 seconds to 85.033 seconds. For the Chinese digit dataset, the correlation between verification time and bit length parallels that observed in the Iris dataset. The verification time similarly escalates alongside augmentations in quantization levels. Hence, it is deducible that improving the quantization level could potentially result in a escalation in verification time, whereas diminishing said quantization level might yield a reduction in verification time. It should be noted, however, that a direct positive correlation between the quantization level and verification time does not invariably hold true. Indeed, in certain instances, augmenting the quantization level has exhibited the outcome of diminishing the verification time. Moreover, the extent of this safety property's range correlates with the amplitude of the oscillations observed.

To assess the impact of quantization on verification outcomes, common quantization levels including 4 bits, 8 bits, 16 bits, and 32 bits are employed. The outcomes of experiments involving varied bit-width quantization and no quantization are tabulated in Table 1. For each quantization level, the procedures are

	Success	**Failure / Timeout**
Non-quantized	5	16
4 bits	21	0
8 bits	9	12
16 bits	7	14
32 bits	6	15

TABLE 1: Verification Outcomes

executed for individual classes and corresponding safety properties. The resultant verification outcomes are systematically documented. To elaborate, the Iris dataset comprises three classifications and three associated safety properties, while the Chinese digit dataset encompasses three classifications and four safety properties. Consequently, each quantization level necessitates verification for 21 program instances ($3 \times 3 + 3 \times 4$). Success denotes instances where all samples within the input domain conform to the stipulated safety property, while Failure signifies the existence of samples in the input domain that contravene the specified safety property. Timeout is indicative of instances wherein verification exceeds the designated time of 10 minutes. Examination of the results reveals that employment of lower-bit quantization tends to skew verification outcomes towards successes. Specifically, utilization of 4-bit quantization results in successful verification for all experimentally employed programs. Conversely, higher-bit quantization yields verification outcomes more closely aligned with the original, non-quantized verification results. Additionally, programs amenable to successful verification predominantly exhibit smaller input domains, whereas those encountering verification failure or timeouts exhibit larger

input domains.

5. Summary

In conclusion, this study employs ESBMC to accomplish the verification of small-size quantized neural networks. Specifically, the neural network is trained employing the Keras framework, and subsequently expressed in the C programming language, with pertinent code excerpts elucidated in Section 3.1. The process of neural network quantization entails the conversion of floating-point numbers to fixed-point numbers, accompanied by the conversion of requisite mathematical operations. The verification of the quantized neural network is undertaken utilizing the ESBMC tool. Subsequent to this, an evaluation of the impact of varied parameters supported by ESBMC on the temporal aspects of verification is conducted, leading to the determination that the majority of these parameters exert little influence on the optimization of the verification process. Within the confines of this experimental endeavor, the parameters amenable to be enabled are "no-bounds-check" and "interval-analysis." A comparative analysis of verification time across distinct SMT solvers reveals that Z3 evinces challenges in completing a substantial proportion of the verification procedures. Meanwhile, the temporal efficiency of the Boolector and Bitwuzla solvers is found to be approximately equivalent, while Yices exhibits the most expeditious verification time. Hence, in the context of this research, Yices emerges as the optimal SMT solver. Lastly, an evaluation into the impact of varying quantization levels on both verification time and outcomes is conducted. The analysis discerns that the extent of quantization is not positively correlated with verification time, characterized by oscillatory increments in verification time as quantization levels ascend. Notably, certain instances manifest a paradoxical reduction in verification time upon elevating the quantization level. Furthermore, diminished quantization levels appear to predispose QNNs towards facile satisfaction with safety properties, indicative of a bias towards success. Conversely, elevated quantization levels align the verification results more closely with those of ANNs.

Several deficiencies persist within this project. The employed dataset in this experiment exhibits simplicity, and concomitantly, the size of the associated neural network is small. This stands in stark contrast to the prevailing trend wherein contemporary datasets tend to possess intricate structures, paralleled by large size neural networks. Consequently, the findings presented by this project lack the depth required for broad generalization. It is imperative to acknowledge that the exclusive utilization of ReLU as the unique activation function is a notable limitation of this experiment. Consequently, the conclusions drawn herein fail to encapsulate the nuanced behaviors that may arise through the adoption of alternative activation functions such as softmax, tanh, and others within neural networks. The further work of this research aims to enhance the generality associated with the verification of quantized neural networks. Specifically, the objective is to successfully accomplish the verification of medium

size neural networks. The initial focus pertains to the selection of widely employed and intricate datasets, exemplified by MNIST. Subsequently, the establishment and training of medium size neural networks tailored to these datasets will be undertaken. The subsequent phase involves the meticulous quantization of activation functions such as softmax and tanh. Subsequent to this quantization, the integration of these activation functions into neural networks will be executed, followed by a comprehensive verification process of the resulting neural networks.

References

[1] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[2] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[3] Chigozie Nwankpa et al. "Activation functions: Comparison of trends in practice and research for deep learning". In: *arXiv preprint arXiv:1811.03378* (2018).

[4] Bekir Karlik and A Vehbi Olgac. "Performance analysis of various activation functions in generalized MLP architectures of neural networks". In: *International Journal of Artificial Intelligence and Expert Systems* 1.4 (2011), pp. 111–122.

[5] Matthew D Zeiler et al. "On rectified linear units for speech processing". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 3517–3521.

[6] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. Atlanta, GA. 2013, p. 3.

[7] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[8] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[9] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[10] Boris T Polyak. "Some methods of speeding up the convergence of iteration methods". In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17.

[11] Yurii Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence O (1/k2)". In: *Dokl. Akad. Nauk. SSSR*. Vol. 269. 3. 1983, p. 543.

[12] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[13] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[14] Itay Hubara et al. "Quantized neural networks: Training neural networks with low precision weights and activations". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[15] Paulius Micikevicius et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017).

[16] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[17] Han Vanholder. "Efficient inference with tensorrt". In: *GPU Technology Conference*. Vol. 1. 2016, p. 2.

[18] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[19] Nadav Rotem et al. "Glow: Graph lowering compiler techniques for neural networks". In: *arXiv preprint arXiv:1805.00907* (2018).

[20] Xidan Song et al. "QNNVerifier: A Tool for Verifying Neural Networks using SMT-Based Model Checking". In: *arXiv preprint arXiv:2111.13110* (2021).

[21] Luiz Sena et al. "Verifying Quantized Neural Networks using SMT-Based Model Checking". In: *arXiv preprint arXiv:2106.05997* (2021).

[22] Daniel Menard, Daniel Chillet, and Olivier Sentieys. "Floating-to-fixed-point conversion for digital signal processors". In: *EURASIP Journal on Advances in Signal Processing* 2006 (2006), pp. 1–19.

[23] Suyog Gupta et al. "Deep learning with limited numerical precision". In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746.

[24] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[25] Thomas A Henzinger, Mathias Lechner, and Đorđe Žikelić. "Scalable verification of quantized neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3787–3795.

[26] Guy Katz et al. "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer. 2017, pp. 97–117.

[27] Markus Nagel et al. "A white paper on neural network quantization". In: *arXiv preprint arXiv:2106.08295* (2021).

[28] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535.

[29] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[30] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held*

as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14. Springer. 2008, pp. 337–340.

[31] Clark Barrett et al. "Cvc4". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer. 2011, pp. 171–177.

[32] Robert Brummayer and Armin Biere. "Boolector: An efficient SMT solver for bit-vectors and arrays". In: *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer. 2009, pp. 174–177.

[33] Bruno Dutertre and Leonardo De Moura. "The yices smt solver". In: *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2.2 (2006), pp. 1–2.

[34] Bowen Alpern and Fred B Schneider. "Recognizing safety and liveness". In: *Distributed computing* 2.3 (1987), pp. 117–126.

[35] Guy Katz et al. "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer. 2017, pp. 97–117.

[36] Guy Katz et al. "The marabou framework for verification and analysis of deep neural networks". In: *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer. 2019, pp. 443–452.

[37] Nina Narodytska et al. "Verifying properties of binarized deep neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[38] Saurabh Karsoliya. "Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture". In: *International Journal of Engineering Trends and Technology* 3.6 (2012), pp. 714–717.

[39] Rory Conlin et al. "Keras2c: A library for converting Keras neural networks to real-time compatible C". In: *Engineering Applications of Artificial Intelligence* 100 (2021), p. 104182.

[40] Mikhail R. Gadelha et al. "ESBMC 5.0: An Industrial-Strength C Model Checker". In: 33^{rd} *ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)*. New York, NY, USA: ACM, 2018, pp. 888–891. DOI: 10.1145/3238147.3240481.