

TDK-dolgozat

Kiss Dorina Hedvig



MATEMATIKAI ÉS INFORMATIKAI INTÉZET

INFORMATIKA KAR

SZÁMÍTÁSTUDOMÁNYI TANSZÉK

Kvantált neurális hálók implementálása formális verifikáció céljából

Szerző:

Kiss Dorina Hedvig

Programtervező
Informatikus BSc

3. évfolyam

Témavezető:

Dr. Kovásznai Gergely

Tanszékvezető,
Egyetemi docens



INSTITUTE OF MATHEMATICS AND INFORMATICS

Implementation of Quantized Neural Networks for the Sake of Verification

Author

Dorina Hedvig Kiss

Computer Science BSc

Supervisor

Gergely Kovasznai, PhD

associate professor

EGER, 2022

Chapter 1

Introduction

Neural Networks (NNs) improved a lot even in the recent years, it became an important and urgent issue to examine the vulnerability of such systems. The aim of this project was to make a neural network, that can be easily verified and still produces multiple values to verify.

For that purpose I implemented a Quantized Neural Network (QNN), which can provide multiple values depending on the power of quantization. I trained the QNN using MNIST dataset, and it earned a really good accuracy. I also managed to extract the weights and the values we needed for the verification, so my supervisor ran the verification using the extracted values.

These accomplishments are discussed in different chapters of this paper.

Chapter 2 contains the necessary information about Neural Networks in general, about the specific ones, such as Binarized Neural Networks (BNNs) and Quantized Neural Networks (QNN), and also the implementation and structure of the latter Neural Network (QNN), as well as the technological foundation.

Chapter 3 covers the topic of parameter extraction from the different layers of QNN, like BATCHNORMALIZATION, and also tells the reader about the encoding of QNN into Boolean cardinality constraints (BCCs).

In chapter 4 the reader can learn about the exact structure of the implemented QNN, the training and the verification results.

The last chapter (ch. 5) contains the conclusion of the project and the further opportunities to continue it.

Chapter 2

Quantized Neural Networks

2.1 Neural Networks in General

At the beginning, in the 1940's artificial intelligence (AI) was no more than a theory based on the structure of human brain, especially neurons and their connection. Later, however, with the development of computer technology, we were already able to implement these theories. “As computers became more advanced in the 1950's, it was finally possible to simulate a hypothetical neural network.” [1]

The appearance of deep-learning neural networks has brought a so-called paradigm shift: a multitude of applications that we thought in the previous decade that were unsolvable without human intelligence. Artificial neural networks now can control autonomous robots, vehicles, or game players more robustly than before.

Neural networks consist of multiple *layers*. These layers are made of *nodes*. It is the place where the computation happens. “A node combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning significance to inputs with regard to the task the algorithm is trying to learn...” [22]

The calculation is a linear combination of the independent variables and their respective weights, and a bias term for each neuron. The neural network equation [23] looks like this:

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}, \tag{2.1}$$

where:

- $\mathbf{y}$ is a vector of output values,
- W is a matrix of weights,
- $\mathbf{x}$ is a vector of independent variables or inputs, and

- **b**: "The bias is used to shift the result of activation function towards the positive or negative side." [24]

This is usually followed by an activation function. These are used "to transform an input signal into an output signal which in turn is fed as input to the next layer in the stack." [13] For instance ReLU (Rectified Linear Unit) is an activation function, which "implies that a neuron will be deactivated only when the output of linear transformation is zero." [?]

Another activation function is SoftMax, which is used in this project. "Softmax function ... can be used for multiclass classification problems. The function, for every data point of all the individual classes, returns the probability." [?]

Then comes error calculation. One of the most common error functions is the Mean Squared Error (MSE) [25], that can be calculated as

$$\text{MSE} = \frac{\sum (y_i - \hat{y}_i)^2}{n},$$

where:

- y_i is the i^{th} observed value,
- $\hat{y}_i$ is the i^{th} predicted value, and
- n is the number of observation.

Error calculation is followed by a calculation called *backpropagation*. "In fact, BP algorithm is a method to monitor learning. It utilizes the methods of mean square error and gradient descent to realize the modification to the connection weight of network. The modification to the connection weight of network is aimed at achieving the minimum error sum of squares." [11] For more information about this calculation see: [11].

Batch normalization can also be inserted into the process, which enables us "to use much higher learning rates and be less careful about initialization, and in some cases eliminates the need for Dropout." [12]

From the theoretical implementation, artificial intelligence has undergone several changes and is one of the most important fields in today's information technology. With its improvement, and the spread of its use vulnerability and its prevention became quite an important topic. Those systems that apply neural networks became so complex in the last few years that examining their security is an essential step before beginning the work with them. It is a proven fact that these kinds of systems can be easily attacked by applying some noise on their input, even if it is such a negligible amount for the human brain. As a result their classification or prediction will not provide the expected outcome.

In the last few years several important papers were published in the topic of verifying the robustness of neural networks (e.g. [4], [5]), but the most of these was about so-called Binarized Neural Networks.

2.1.1 Binarized Neural Networks

Binarized Neural Networks (BNNs) are similar to the regular feedforward neural networks, except that their weights and activations should be constrained only to 2 values, to either -1 or 1 . [2]

This constraint in the number of values helps to reduce not only memory size, but also processing time. “At training-time the binary weights and activations are used for computing the parameters gradients. During the forward pass, BNNs drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations, which is expected to substantially improve power-efficiency.” [3]

Figure 2.1 shows a common construction of a BNN. “Some approaches describe the structure of a BNN in terms of sequential composition of blocks of layers rather than individual layers. While the blocks can produce real-typed intermediate values, each of them takes a binary input vector and outputs a binary vector, except for the output block.” [7] Each internal block contains three layers: linear transformation with binarized weights (BLIN), batch normalization (BN), and binarization (BIN). The output block provides the classification decision for a given binary input vector. It consists of two layers: a BLIN that outputs a vector of integers followed by an ARGMAX layer. [7]

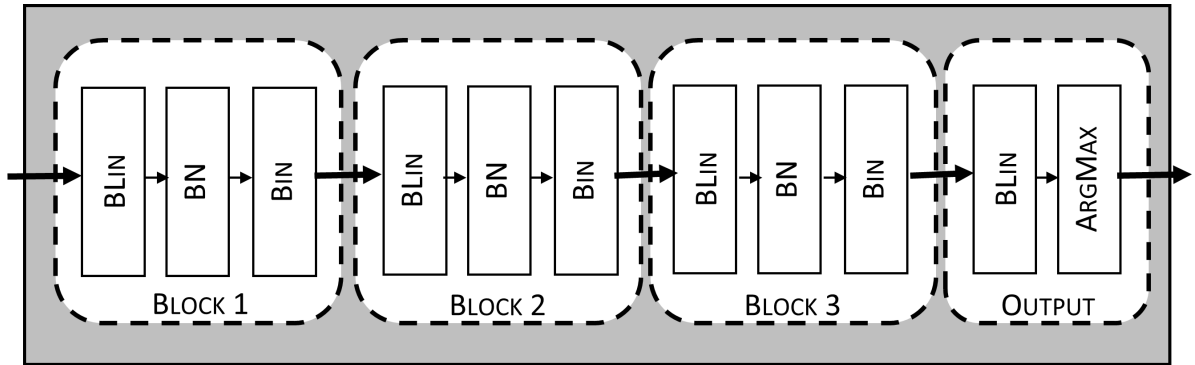


Figure 2.1: A schematic view of a Binarized Neural Network. [7]

Gradient computation is, however, a bit problematic. It is to minimize cost function, but since a BNN produces only 2 values, computing this from the derivative of the function is not possible, because the function will have points of discontinuity. “The gradient $\nabla f(\omega)$ in training activation quantized DNN is almost everywhere (a.e.) zero, which makes the standard back-propagation inapplicable. The arguably most effective way around this

issue is nothing but to construct a non-trivial search direction by properly modifying the chain rule. Specifically, one can replace the a.e. zero derivative of quantized activation function composited in the chain rule with a related surrogate. This proxy derivative used in the backward pass only is referred as the straight-through estimator (STE) ... proposed an alternative approach based on stochastic neurons.” [9] That explores the space of parameters in small and noisy steps, and that noise is averaged out by the stochastic gradient contributions accumulated in each weight. [2]

Straight-Through Estimator (STE), however, is to estimate the gradients of functions ignoring derivatives. It was observed that “the fastest training was obtained when using the *straight-through estimator*”. [2]

2.2 Implementing Quantized Neural Networks

Floating-point parameters in neural networks provide us a wide variety of values, which makes the search space large and, thus, verification hard. On the other hand, BNNs can be successfully verified, but they provide only 2 kinds of values that can be either -1 or 1 . Taking this into consideration, we decided to implement Quantized Neural Networks (QNNs) that are able to provide several but still a finite number of values due to quantization.

QNNs have a similar structure to BNNs. QNNs are also made up of blocks of layers as Figure 2.2 shows. The first block that should be mentioned is the *preprocessing* block that consists of a *batch normalization* layer and an *activation* function. The role of this block is to produce quantized values to work with from the very beginning.

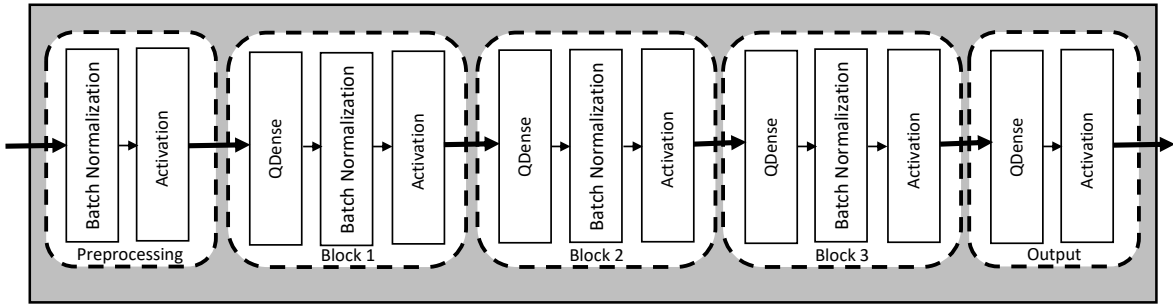


Figure 2.2: A schematic view of a QNN.

The preprocessing block is followed by one or more internal blocks. Each of them contains three layers: a *linear transformation* with quantized weights (in our case the QDense layer), a *batch normalization*, and then an *activation*.

The output block consist of only two layers: a *linear transformation* with quantized weights and an *activation*.

Our implementation of QNNs is based on a BNN implementation¹ from GitHub.

2.2.1 Technological Foundation

The base of the project was chosen to be *Keras*. This is an open-source machine learning library, that aims to be user-friendly and flexible, and it was initially developed as part of the research effort of project ONEIROS. [18] “Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation.” [18]

So *Keras* is built on top of *TensorFlow*. ”TensorFlow is a Python-friendly open source library for numerical computation that makes machine learning and developing neural networks faster and easier.” [17] It combines such key abilities [18] as follows:

- Efficiently executing low-level tensor operations on CPU, GPU, or TPU.
- Computing the gradient of arbitrary differentiable expressions.
- Scaling computation to many devices, such as clusters of hundreds of GPUs.
- Exporting programs (“graphs”) to external runtimes such as servers, browsers, mobile and embedded devices.

2.2.2 Quantization

At first, let us specify a *quantization bit-width* $\mathbf{bw} \in \mathbb{N}$, which makes quantization be applied along $2^{\mathbf{bw}}$ threshold values. The threshold values $Q_q^{\mathbf{bw}}$ can be calculated as follows:

$$Q_q^{\mathbf{bw}} = \frac{2q - 1}{2^{\mathbf{bw}}} - 1, \text{ where } q = 1, \dots, 2^{\mathbf{bw}}.$$

For instance, if $\mathbf{bw} = 2$ then

$$Q_1^2 = -0.75, Q_2^2 = -0.25, Q_3^2 = 0.25, Q_4^2 = 0.75.$$

A value $x \in \mathbb{R}$ is quantized as follows:

$$Q'(x, \mathbf{bw}) = \begin{cases} 0, & \text{if } x < Q_1^{\mathbf{bw}}, \\ 2^{\mathbf{bw}}, & \text{if } x \geq Q_{2^{\mathbf{bw}}}^{\mathbf{bw}}, \\ q, & \text{if } Q_q^{\mathbf{bw}} \leq x < Q_{q+1}^{\mathbf{bw}}, \text{ where } 1 \leq q < 2^{\mathbf{bw}} \end{cases}$$

¹<https://github.com/Haosam/Binary-Neural-Network-Keras>

and then

$$Q(x, \text{bw}) = \frac{Q'(x, \text{bw})}{2^{\text{bw}-1}}$$

provides the quantized value. From that, the following formula can be built, which was used in our implementation of QNNs:

$$Q(x, \text{bw}) = \frac{\text{Round}(2^{\text{bw}-1}x)}{2^{\text{bw}-1}}, \quad (2.2)$$

where $\text{Round}(\cdot)$ applies rounding to the closest integer.

We implemented a function to quantize not just a single real value, but even a matrix of such numbers. Using the formula (2.2), we implemented a quantization function as it can be seen in Listing 2.1, where we call the `round` function from the Keras backend.

```

1  def round_quantize(x, bw):
2      q_pow = 2**(bw-1)
3      numerator = q_pow * x
4      numerator = numerator + K.stop_gradient(K.round(numerator) - numerator)
5      return numerator / q_pow

```

Listing 2.1: Quantization function.

Figure 4.1 shows an example of how weights in a `Dense` layer get quantized. In this case, the bit-width of quantization is selected to be `bw = 2`, and the values are sorted for easier interpretation.

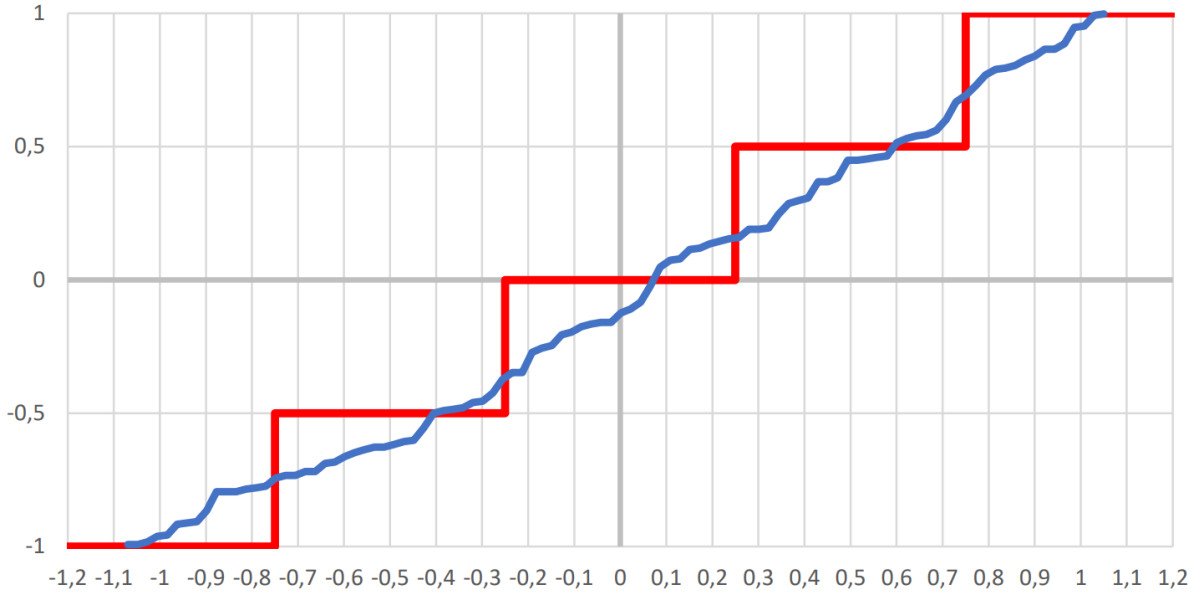


Figure 2.3: Example of quantization with bit-width `bw = 2`.

The quantization function has several points of discontinuity. Although this seems an unimportant detail, that should be taken into consideration since in case of discontinuous

functions computing the derivative function is impossible. Therefore no gradient can be computed. As Keras was the chosen library to implement QNNs, `stop_gradient` in the Keras backend can be used to handle a function of this kind, as it can be seen in Listing 2.1, because it disables gradient calculation. Such a solution for gradient computation is a possible way to implement a so-called Straight-Through Estimator (STE).

2.2.3 QDense Layer

Now that a function for quantization has been implemented, it can be used inside a layer to make a neural network to compute quantized values. Therefore, we use the method in Listing 2.1 inside our `QDense` layer, which is derived from the `Dense` class of *Keras*. `QDense` expects for the same list of parameters as the parent class, plus another one that stands for the quantization bit-width, as it can be seen in Listing 2.2.

```

1 class QDense(Dense):
2     def __init__(self, units, bit_width, H=1., kernel_lr_multiplier='Glorot',
3         bias_lr_multiplier=None, **kwargs):
4         super(QDense, self).__init__(units, **kwargs)

```

Listing 2.2: Parameters for the `QDense` layer.

Later on, the bit-width will be passed to the `round_quantize` method, which is called inside the `call` method of `QDense`:

```

1 def call(self, inputs):
2     output = K.dot(inputs, round_quantize(self.kernel, self.bit_width))
3     if self.use_bias:
4         return K.bias_add(output, self.bias)
5     if self.activation is not None:
6         return self.activation(output)

```

Listing 2.3: The `call` method of the `QDense` layer.

Weight and bias values are stored as floating-point numbers inside the current layer, but the subsequent layer will receive them after quantization.

Chapter 3

Processing Trained QNNs for Verification

The formal verification of a QNN relies on the parameter values that have been set inside the QNN during the training process. Therefore, those parameters must be extracted from all the layers that build up the QNN. Sections 3.1 and 3.2 show how to extract them from `QDense` and batch normalization layers.

Those parameters can be used for expressing all the calculations and activations inside a QNN as a set of Boolean Cardinality Constraints, as Sections 3.4 and 3.5 illustrate.

3.1 Parameters from QDense Layers

The first component that should be mentioned is the `Dense` class in *Keras*. This class implements the equation (2.1). This is the class which `QDense` inherits from. Listing 3.1 shows a section of the source code¹ of the class `Dense`. As it can be seen, there are two implementations of equation (2.1) based on the rank.

```
1 def call(self, inputs):
2     #...
3     rank = inputs.shape.rank
4     if rank == 2 or rank is None:
5         if isinstance(inputs, tf.SparseTensor):
6             inputs, _ = tf.sparse.fill_empty_rows(inputs, 0)
7             ids = tf.SparseTensor(
8                 indices=inputs.indices,
9                 values=inputs.indices[:, 1],
10                dense_shape=inputs.dense_shape,
11            )
```

¹Source: <https://github.com/keras-team/keras/blob/v2.11.0/keras/layers/core/dense.py>

```

12         weights = inputs
13         outputs = tf.nn.embedding_lookup_sparse(
14             self.kernel, ids, weights, combiner="sum"
15         )
16     else:
17         outputs = tf.matmul(a=inputs, b=self.kernel)
18     else:
19         outputs = tf.tensordot(inputs, self.kernel, [[rank - 1], [0]])
20         if not tf.executing_eagerly():
21             shape = inputs.shape.as_list()
22             output_shape = shape[:-1] + [self.kernel.shape[-1]]
23             outputs.set_shape(output_shape)
24         #...

```

Listing 3.1: Implementation of (2.1) inside the class `Dense`.

If the rank is 2 and the input is sparse, a method `embedding_lookup_sparse` is used. It is similar to `matmul` that is used in when the inputs is not sparse, but more efficient for large sparse input tensors². “If the input to the layer has a rank greater than 2, then `Dense` computes the dot product between the inputs and the kernel along the last axis of the inputs and axis 0 of the kernel (using `tf.tensordot`).” [19]

The layer has two parameters to extract:

- *kernel* is the weight matrix,
- *bias* is the bias vector.

Naming the individual layers is also important, because we can refer to each layer and its stored parameters later on.

3.2 Parameters from Batch Normalization Layers

The layer type `BatchNormalization` normalizes the output during training and “...for each channel being normalized, the layer returns $\text{gamma} * (\text{batch} - \text{mean}(\text{batch})) / \sqrt{\text{var}(\text{batch}) + \text{epsilon}} + \text{beta}$...” It also normalizes the output during inference, “it returns $\text{gamma} * (\text{batch} - \text{self.moving_mean}) / \sqrt{\text{self.moving_var} + \text{epsilon}} + \text{beta}$.” [21]

These parameters [21] represent the following:

- *epsilon* is small constant (configurable as part of the constructor arguments).
- *gamma* is a learned scaling factor (initialized as 1), which can be disabled by passing *scale=False* to the constructor.

²Mentioned in the source code row 213

- *beta* is a learned offset factor (initialized as 0), which can be disabled by passing *center=False* to the constructor.
- *self.moving_mean* and *self.moving_var* are non-trainable variables that are updated each time the layer is called in training mode.

3.3 Extracting Parameters

The parameters of our `QDense` and `BatchNormalization` layers can be extracted using the `save_weights` method of the class `Sequential`. This method saves the those values into the given folder with the given name and format, which is, in our case, *H5*. These files are perfect to store the weight matrices and also the parameters from `BatchNormalization`. “An H5 is one of the Hierarchical Data Formats (HDF) used to store large amount of data. It is used to store large amount of data in the form of multidimensional arrays. The format is primarily used to store scientific data that is well-organized for quick retrieval and analysis.” [20]

However, the values `save_weights` saves into the given file are not quantized, since the layers store the actual floating-point number they calculated. The next layer, however, will receive quantized values to work with, because in the `call` method of `QDense` the values get quantized, as Listing 2.3 shows.

Thus, if we want to work with quantized values, we should quantize the weights stored in the *H5* file. Reading weights of different layers from the *H5* file is done as in Listing 3.2. As it can be seen, we refer the stored weights using the name of the layer. Then, the weights get quantized using the method `round_quantize` in Listing 3.2, in row 3.

```

1 model.save_weights(datafile)
2 with h5py.File(datafile, 'r+') as hdf:
3     kernel0 = round_quantize(np.array(hdf.get('/qlayer0/qlayer0/kernel:0')),
4     1)
5     bias0 = np.array(hdf.get('/qlayer0/qlayer0/bias:0'))

```

Listing 3.2: Reading the parameters of the `QDense` layer `qlayer0` from an *H5* file.

We can also read the values of a `BatchNormalization` layer from the *H5* file. This can be similarly done to referring the weights, but beside the name of the layer we should give the name of the parameters as the Listing 3.3 shows.

```

1 with h5py.File(datafile, 'r+') as hdf:
2     variance0 = np.array(hdf.get('/bnlayer0/bnlayer0/moving_variance:0'))
3     mean0 = np.array(hdf.get('/bnlayer0/bnlayer0/moving_mean:0'))
4     beta0 = np.array(hdf.get('/bnlayer0/bnlayer0/beta:0'))

```

```
5 gamma0 = np.array(hdf.get('/bnlayer0/bnlayer0/gamma:0'))
```

Listing 3.3: Extracing the parameters of the `BatchNormalization` layer `bnlayer0` from an `H5` file.

3.4 Encoding QNNs into BCCs

The calculations and decisions applied inside a neural network can be encoded as a set of numerical constraints, which can be fed into a constraint solver for verifying certain properties over the neural network. In case of a QNN, those numerical constraints allow a compact encoding in Boolean logic, enabling verification approaches based on SAT or SMT solving, or 0-1 Integer Linear Programming. An encoding of a QNN into a set of Boolean Cardinality Constraints is proposed in [7].

A *Boolean Cardinality Constraint (BCC)* is defined as an expression $\sum_{i=1}^n l_i \circ_{\text{rel}} c$, where $l_1, \dots, l_n$ are Boolean literals, $\circ_{\text{rel}} \in \{\geq, \leq, >, <, =\}$, and $c \in \mathbb{N}$ is a constant where $0 \leq c \leq n$. A *Boolean literal* is a Boolean variable x or its negation $\neg x$.

According to [7], an internal block of a QNN can be encoded as a set of reified BCCs of the following form:

$$\sum_{j \in J_i^+} \mathbf{1} \cdot \mathbf{x}_j^{\text{bl}} + \sum_{j \in J_i^-} \mathbf{1} \cdot \neg \mathbf{x}_j^{\text{bl}} \circ_{\text{rel}} D_{i,q} \Leftrightarrow o_{i,q}^{\text{bl}}, \quad (3.1)$$

where $\mathbf{x}_j^{\text{bl}}$ represents the j^{th} input value as bit-vector, and $o_{i,q}^{\text{bl}}$ represents the q^{th} bit of the output value as a bit-vector. All bits are represented by individual Boolean variables. About further notations in (3.1):

- J_i^+ and J_i^- are collections of input indices according to the sign of the corresponding weights of the `QDense` layer;
- $D_{i,q}$ is an integer constant that can be computed from the parameters of the `QDense` layer (weight matrix and bias vector) and those of the batch normalization layer (γ , β , moving mean and moving variance).

For the precise definition of those notations and for further details see [7].

3.5 Generating Cardinality Constraints

We implemented the previously described QNN encoding in Python, where we utilized all the previously extracted parameters from the QNN's layers. [7] Listing 3.4 shows the implementation of generating BCCs for an internal block, according to (3.1).

```

1 lits = []
2 for j in range(len(inputVars[k])):
3     if A[k][i][j] > 0:
4         lits.extend(inputVars[k][j])
5     elif A[k][i][j] < 0:
6         lits.extend([solver.negateVar(x) for x in inputVars[k][j]])
7
8 solver.addConstraint(Constraint(
9     lits = lits,
10    relation = Relations.GreaterOrEqual if gamma[k][i] > 0 else Relations.
        LessOrEqual,
11    bounds = D[k][i],
12    resLits = inputVars[k + 1][i]
13 ))

```

Listing 3.4: Generating BCCs for an internal block.

Note, for instance, that the sign of the layer parameter **gamma** determines which the relational operator to use inside the BCC. Similarly, the sign of the elements of the weight matrix **A** determines if the layer’s input variables have to be negated. The resulting value of the BCCs are “saved” into the input variables of the subsequent QNN block, as it is set by the **resLits** parameter.

Chapter 4

Experiments

We ran experiments with a 3-blocked QNN that we trained over the MNIST dataset. Further technical details will be given in Sections 4.1 and 4.2. Then, we extracted the parameters of all the layers in order to verify the trained QNN. Then, we used our Python implementation of the encoding into BCCs and, finally, we were able to run a cardinality solver to do the actual verification. Section 4.3 shows the results when the cardinality solver MINICARD is used.

4.1 Structure of the QNN

As mentioned in Section 2.2, a QNN consists of blocks of layers, as shown in Figure 2.2. The QNN we implemented for our experiments had the exact same structure, as Listing 4.1 shows.

```
1 model = Sequential()
2 model.add(BatchNormalization(input_shape=(28*28,), name='bnlayer0', epsilon
    =epsilon))
3 model.add(Activation(lambda x: round_quantize(x, quantizationPow)))
4 model.add(QDense(200, 1, input_shape=(28*28,), H=H,
5     kernel_lr_multiplier=kernel_lr_multiplier, use_bias=
        use_bias, name='qlayer1'))
6 model.add(BatchNormalization(name='bnlayer1', epsilon=epsilon))
7 model.add(Activation(lambda x: round_quantize(x, quantizationPow)))
8 model.add(QDense(100, 1, H=H,
9     kernel_lr_multiplier=kernel_lr_multiplier, use_bias=
        use_bias, name='qlayer2'))
10 model.add(BatchNormalization(name='bnlayer2', epsilon=epsilon))
11 model.add(Activation(lambda x: round_quantize(x, quantizationPow)))
12 model.add(QDense(100, 1, H=H,
```

```

13         kernel_lr_multiplier=kernel_lr_multiplier , use_bias=
           use_bias , name='qlayer3'))
14 model.add(BatchNormalization(name='bnlayer3' , epsilon=epsilon))
15 model.add(Activation(lambda x: round_quantize(x, quantizationPow)))
16 model.add(QDense(nb_classes , 1, H=H,
17                 kernel_lr_multiplier=kernel_lr_multiplier , use_bias=
           use_bias , name='qlayer4'))
18 model.add(Activation(tf.keras.activations.softmax))
19 model.build()

```

Listing 4.1: The code of the implemented QNN

The code shows not only the structure of the QNN, but also the exact number of neurons in the `QDense` layers, which is 200, 100 and 100, respectively. Furthermore, the code shows the quantization bit-width, which is 1 for the `QDense` layers and `quantizationPow=2` for the `Activation` layers. In all the activation layers, we call the method `round_quantize`, except for the activation of the output block, where `softmax` provides a single value as output.

4.2 Experiments with the MNIST Dataset

Identifying numbers is an easy task for the human brain, even when we talk about handwritten, not uniform numbers. But for a neural network it can be hard to recognise them.

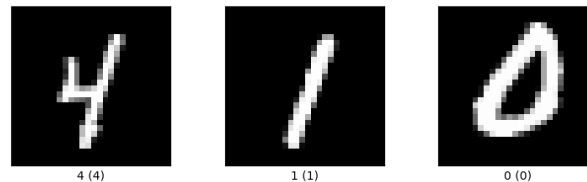


Figure 4.1: Elements of MNIST dataset
Source: <https://www.tensorflow.org/datasets/catalog/mnist>

“The MNIST database of handwritten digits contains 60,000 training examples and 10,000 testing examples, which are 28 * 28 images. All of digits have already been size-normalized and preprocessed and formatted (LeCun et al., 1998). The four files provided on the website are used in the training and testing for neural networks.” [26]

Using the MNIST dataset, we trained our QNN through 10 epochs, and it earned an approximately 91% accuracy, which is shown in Figure 4.2. Taking into consideration that all the weights and inputs are quantized, it is a quite good result.

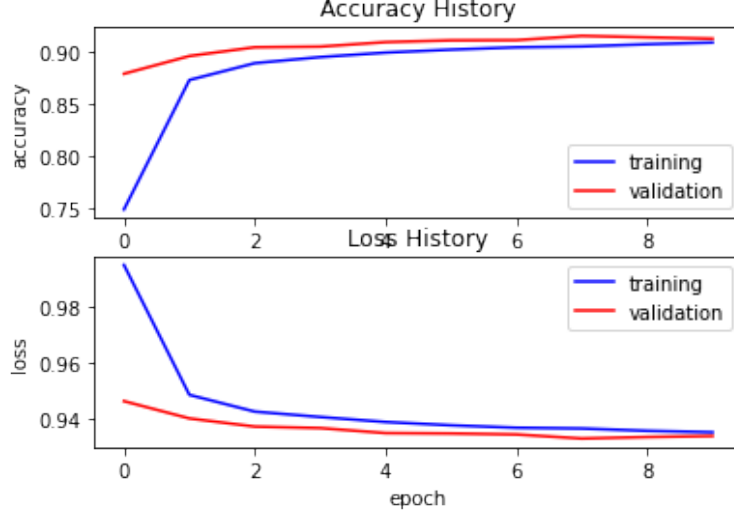


Figure 4.2: Results of training the QNN using MNIST dataset

4.3 Verification of the Trained QNN

“The QNN blocks are transformed into Boolean constraints, which can now fed into a constraint solver, for the sake of formal verification.” [7] The BCC encoding of QNN blocks are implemented in Python inside our tool VERBiNE, that is able to execute different kind of solvers (SAT, SMT, ILP), including cardinality solvers, via Python packages such as PySAT [15], PySMT [16] or Google’s OR-Tools [27].

In our experiments, we run a cardinality solver called MINICARD on the constraints we got. “MiniCARD is a cardinality solver based on MiniSAT... MiniCARD handles cardinality constraints natively, using the same efficient data structures and techniques MiniSAT uses for clauses, giving it much better performance on cardinality constraints than CNF encodings of those constraints passed to a typical SAT solver.” [14]

The use case for our experiments was to verify the adversarial robustness of the trained QNN, meaning that it might have misclassified inputs if we allowed to add perturbation in the range $[-\epsilon, \epsilon]$ to individual input values. For this, we randomly picked 20 images that were correctly classified by the network and then we experimented with three different maximum perturbation values by varying $\epsilon \in \{1, 3, 5\}$. Our experiments were run on Intel i5-7200U 2.50 GHz CPU (2 cores, 4 threads) with 8 GB memory. The time limit was set to 1200 seconds. Figure 4.3 shows the results of our experiments.

VERBiNE, when applying MINICARD as the underlying cardinality solver, was able to complete the verification task under the time limit, except for one of the MNIST images for $\epsilon = 3$. For all the other cases, VERBiNE was able to prove the resulting set of BCCs to be satisfiable, meaning that the trained QNN was not robust on the input images against the given perturbations. Furthermore, VERBiNE was able to generate

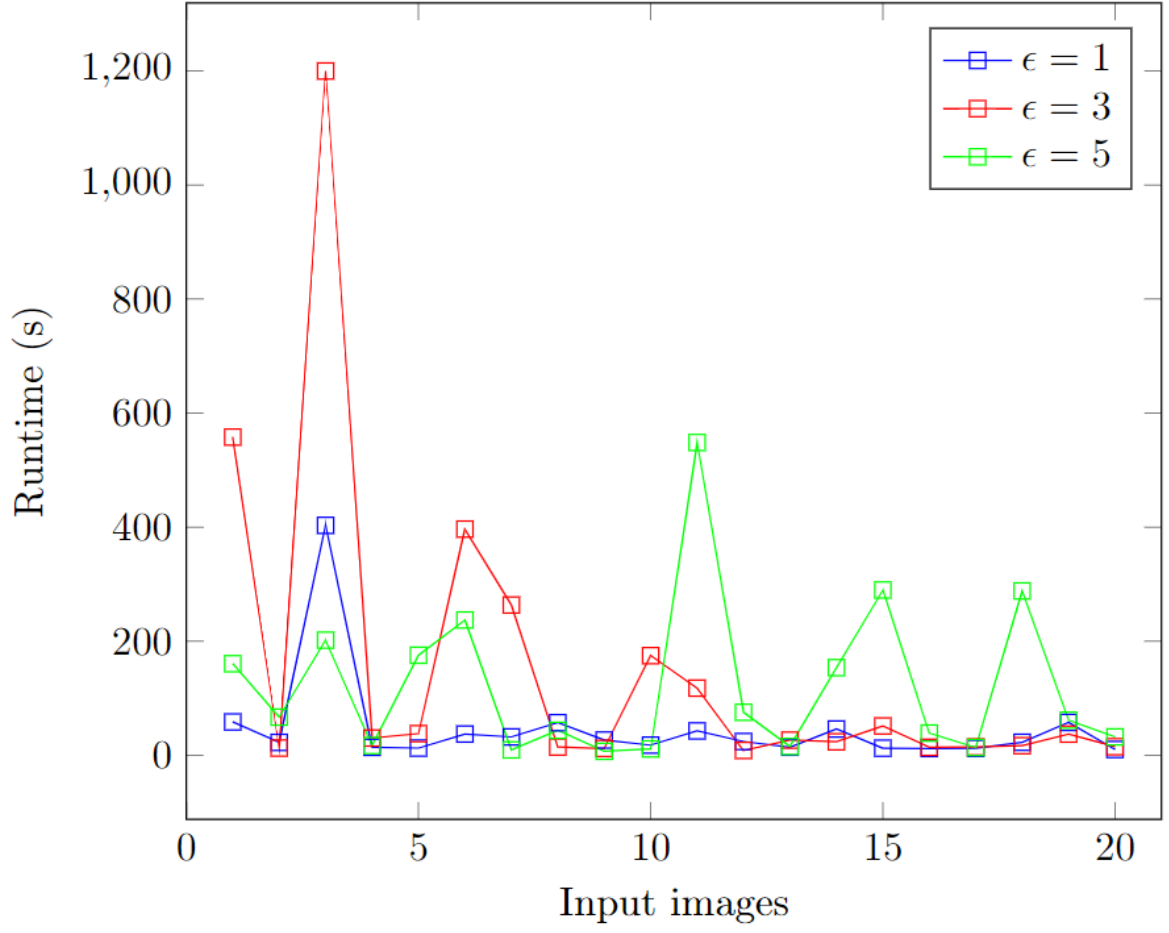


Figure 4.3: Runtimes of VERBINE when running MINICARD on MNIST images with maximum perturbation $\epsilon \in \{1, 3, 5\}$.

the corresponding perturbation matrices.

Chapter 5

Conclusion

To sum up, the implemented QNN works as expected when training using MNIST dataset. It earns a pretty good accuracy and still operates with quantized values we wanted. We also managed to extract the weights and the values from batch normalization, and using these, the QNN could be verified.

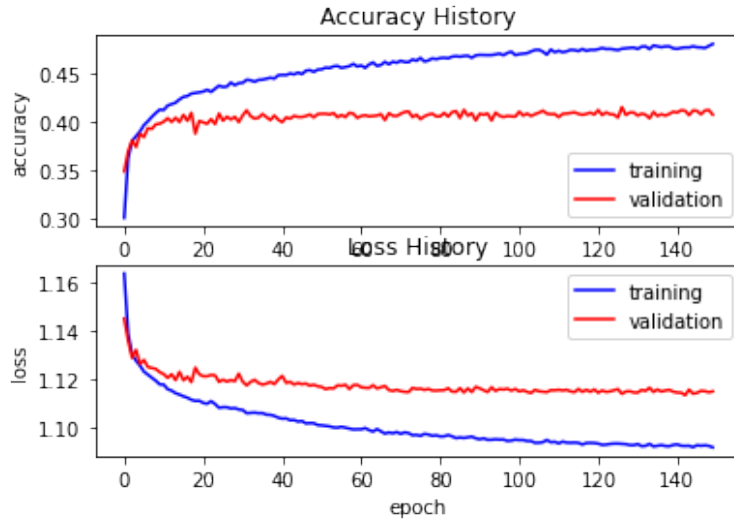


Figure 5.1: Accuracy and loss of QNN trained using CIFAR-10 dataset

Although the training of the QNN went well using MNIST, the experiments with CIFAR-10 were not so promising. It earned much less accuracy as figure 5.1 shows, even when it was trained through more than a hundred epochs, so it is a potential way to continue our work with the QNN.

Bibliography

- [1] Caroline Clabaugh, Dave Myszewski and Jimmy Pang, "Neural Networks", <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html>
- [2] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv and Yoshua Bengio, "Binarized Neural Networks", *Advances in Neural Information Processing Systems* 29, pp. 4107–4115, 2016.
- [3] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv and Yoshua Bengio, "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1", *arXiv:1602.02830*, 2016.
- [4] K. Jia and M. Rinard, "Efficient Exact Verification of Binarized Neural Networks," *Advances in Neural Information Processing Systems*, pp. 1782–1795, 2020.
- [5] G. Kovásznai, K. Gajdár and N. Narodytska, "Portfolio solver for verifying Binarized Neural Networks," *Annales Mathematicae et Informaticae*, vol. 53, pp. 185–202, <https://doi.org/10.33039/ami.2021.03.007>, 2021.
- [6] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2017.
- [7] G. Kovásznai, D. H. Kiss, P. Mlinkó, "Formal Verification for Quantized Neural Networks," *Annales Mathematicae et Informaticae*, 2022. (under revision)
- [8] Kevin Clark, "Computing Neural Network Gradients".
<https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf>
- [9] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, Jack Xin, "Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets", *International Conference on Learning Representations (ICLR)*, 2019.
<https://arxiv.org/pdf/1903.05662.pdf>
- [10] Sun-Chong Wang, "Artificial Neural Network". *Interdisciplinary Computing in Java Programming*, Springer US, pp. 81–100, 2003.
https://doi.org/10.1007/978-1-4615-0377-4_5.

- [11] Li, J., Cheng, Jh., Shi, Jy., Huang, F. , "Brief Introduction of Back Propagation (BP) Neural Network Algorithm and Its Improvement", Advances in Computer Science and Information Engineering, Advances in Intelligent and Soft Computing, vol 169, 2012.
https://doi.org/10.1007/978-3-642-30223-7_87
- [12] Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", Proceedings of Machine Learning Research, vol. 37, pp. 448–456, 2015.
<http://proceedings.mlr.press/v37/ioffe15.html>
- [13] Siddharth Sharma, Simone Sharma, Anidhya Athaiya, "Activation functions in Neural Networks", International Journal of Engineering Applied Sciences and Technology, vol. 4(12), pp. 310–316, 2020.
- [14] M. H. Liffiton, J. C. Maglalat, "More Expressive Constraints for Free", Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 7317, Lecture Notes in Computer Science, Springer, 2012, pp. 485—486.
- [15] A. Ignatiev, A. Morgado, J. Marques-Silva, "PySAT: A Python Toolkit for Prototyping with SAT Oracles", Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 10929, Lecture Notes in Computer Science, Springer, 2018, pp. 428—437.
- [16] M. Gario, A. Micheli, "PySMT: A Solver-Agnostic Library for Fast Prototyping of SMT-based Algorithms", International Workshop on Satisfiability Modulo Theories (SMT), 2015.
- [17] Serdar Yegulalp, "What is TensorFlow? The machine learning library explained", 2022.
<https://www.infoworld.com/article/3278008>
- [18] Documentation of Keras, 2022.
<https://keras.io/about/>
- [19] Documentation of the Dense layer in Keras, 2022.
https://keras.io/api/layers/core_layers/dense/
- [20] Documentation of the HDF file format, 2022.
<https://docs.fileformat.com/misc/h5/>

- [21] Documentation of the BatchNormalization layer in Keras, 2022.
https://keras.io/api/layers/normalization_layers/batch_normalization/
- [22] Chris Nicholson, "A Beginner's Guide to Important Topics in AI, Machine Learning, and Deep Learning", 2022.
<https://wiki.pathmind.com/neural-network>
- [23] Neha Seth, "Estimation of Neurons and Forward Propagation in Neural Net", 2021.
<https://www.analyticsvidhya.com/blog/2021/04/estimation-of-neurons-and-forward-p>
- [24] Farhad Malik, "Neural Networks Bias And Weights", 2019.
<https://medium.com/fintechexplained/10b53e6285da>
- [25] Jim Frost, "Mean Squared Error (MSE)", 2022.
<https://statisticsbyjim.com/regression/mean-squared-error-mse/>
- [26] Huimin Wu, "CNN-Based Recognition of Handwritten Digits in MNIST Database", 2018. http://users.cecs.anu.edu.au/~Tom.Gedeon/conf/ABCs2018/paper/ABCs2018_paper_104.pdf
- [27] L. Perron, V. Furnon, "OR-Tools", version 9.5, Google, <https://developers.google.com/optimization/>.