Project Report

Prabhroop Singh
University of Alberta

Abstract
In this project, our main focus was to improve the CNN model's training time using log quantization and using taylor series approximation and lookup tables to find the log quantized values. We then used the obtained log values in the training process on the inputs as well as after each layer in order to enhance the computational efficiency. The proposed model was also compared with two similar classifiers to compare and contrast their accuracies and training times. Furthermore a simplified vhdl implementation was also designed to simulate the working of the log quantization method.

Keywords: CNN, Hardware Efficiency, Log Quantization, Taylor Series, Lookup Tables

1. Introduction

The Convolutional Neural Network (CNN) has demonstrated outstanding effectiveness in a wide range of tasks, ranging from image recognition, object detection, natural language processing, medical imaging, video analysis to autonomous driving applications. But the computational demands of these networks have significantly increased, this substantial computing burden presents difficulty for these systems in real-time applications when predictions need to be made with the least amount of delay.(Zhao et al., 2018). Also with the increasing complexities of the models working on a wide variety of datasets with considerable variance and diversity in their feature sets, the training times of the models have also increased. Therefore in this project we aim to address the challenge of high training time by using log quantization method with taylor series approximation and lookup tables to optimize CNN architectures for efficient training times. Specifically, we focus on reducing computational complexity while sacrificing negligible accuracy.

2. Background research

The ideas in this project were inspired by Miyashita, D., Lee, E. H., & Murmann, B. (2016) in Convolutional neural networks using logarithmic data representation and also the concepts of lookup tables studied in ECE 511 lectures.In Convolutional Neural Networks using Logarithmic Data Representation, the authors propose two methods to enhance the training process of the CNN models.

The first method involves using an approach to optimize computations by using logarithmic quantization. Initially, a logarithmic quantization process is applied on one of the operands to transform it into a representation that corresponds to a power of 2. After this step, the equivalent computation is carried out using a bit-shifting operation. This transformation eliminates the need for resource-intensive multiplication operations, replacing them with a far less computationally expensive bitwise operation. By implementing this method the overall complexity of training is significantly reduced. The multiplication and shifting operation is depicted as follows:

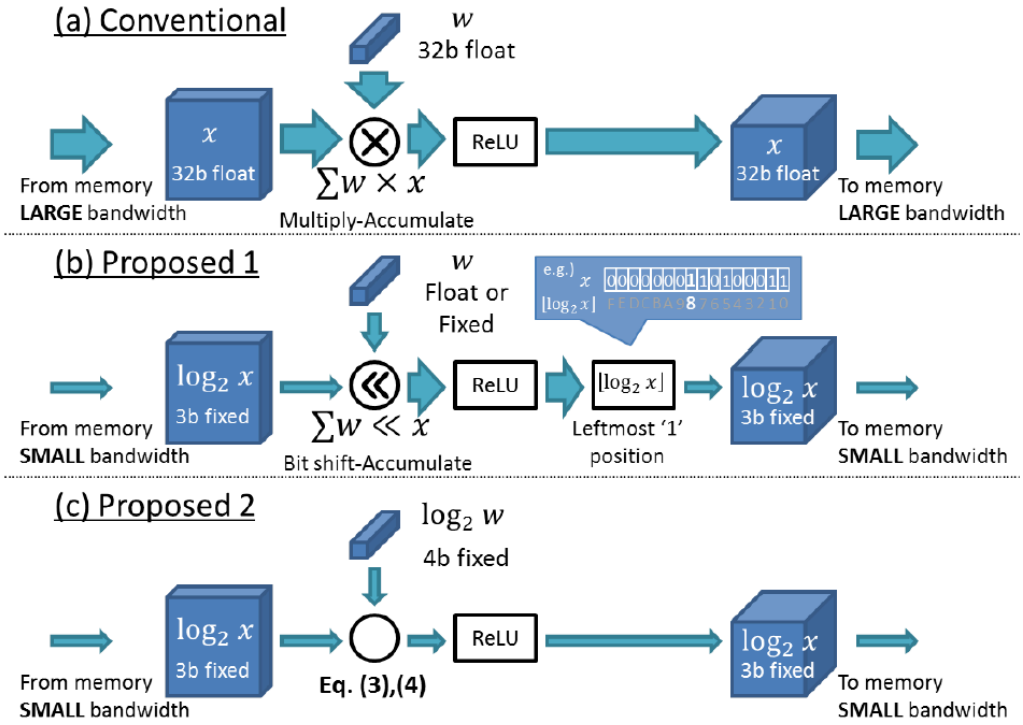

Fig 1.1 Concept and motivation of this study. Adapted from "Convolutional Neural Networks Using Logarithmic Data Representation," by D. Miyashita, E. H. Lee, & B. Murmann, 2016, *arXiv preprint arXiv:1603.01025*. Copyright 2016 by the authors.

$$
\begin{aligned}
w^T x &\simeq \sum_{i=1}^{n} w_i \times 2^{\tilde{x}_i} \\
&= \sum_{i=1}^{n} \mathrm{Bitshift}(w_i, \tilde{x}_i),
\end{aligned}
$$

A second method is also proposed by the authors. In this approach, log quantization is applied on both inputs. After quantizing the inputs logarithmically, bitshift operation operation is used to calculate the equivalent final result.

$$
\begin{aligned}
w^T x &\simeq \sum_{i=1}^{n} 2^{\mathrm{Quantize}(\log_2(w_i))+\mathrm{Quantize}(\log_2(x_i))} \\
&= \sum_{i=1}^{n} \mathrm{Bitshift}(1, \tilde{w}_i + \tilde{x}_i),
\end{aligned}
$$

The above methods are used to replace the multiplication accumulation to Bit-Shift accumulations.

The main reason for working in the logarithmic domain instead of linear domain is that we can convert the numbers into the log domain (taking the log of numbers) and then perform the operations. This reduces the need for large bit widths because the logarithmic scale compresses large numbers. According to Miyashita et al., 2016, using either linear or log domains have their pros and cons but their proposed is efficient because of the simpler calculations at a hardware level.

We have also utilized the general concept of lookup tables. Lookup tables can be used to store certain precomputed values which can be stored and reused whenever required in the program.

3. Methodology

We used Taylor series approximation with integrated lookup tables to find approximate log quantized values. The method used is detailed in Fig 2.1.

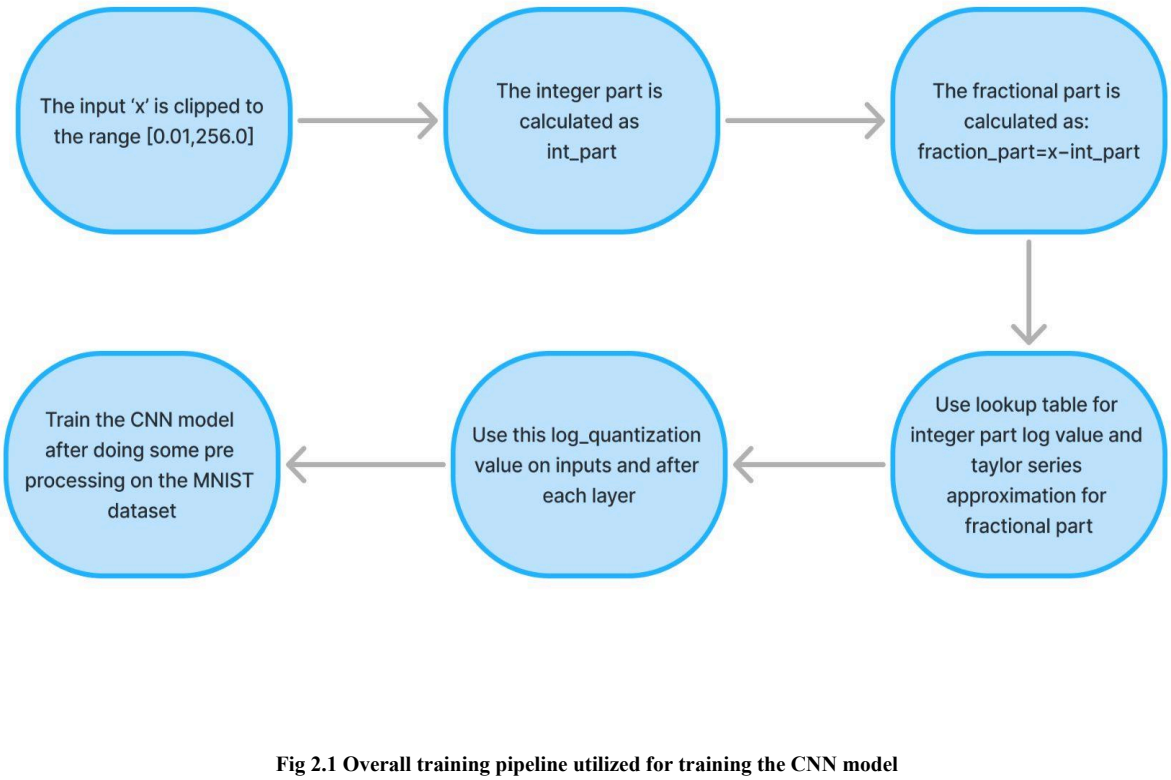

Fig 2.1 Overall training pipeline utilized for training the CNN model

The input 'x' is first of all clipped to be in the range 0.01 to 256 to ensure stability by avoiding any negative values and to maintain simplicity in this code. Then the integer part and fractional part calculated as follows:

```
x_clipped = tf.clip_by_value(x, 0.01, 256.0)
int_part = tf.cast(tf.math.floor(x_clipped), tf.int32)
fraction_part = x_clipped - tf.cast(int_part, tf.float32)
```

Then we compute the logarithmic values of ‘int_part’ and ‘fract_part’ separately.

To calculate the log value of the ‘int_part’ a lookup table was used. The lookup table was generated as follows:

```
log2_lookup_np = np.log2(np.arange(1, 257))
log2_lookup = tf.constant(log2_lookup_np, dtype=tf.float32)
```

To calculate the fractional part’s log value, taylor series approximation was used:-

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

But to approximate the result of small ‘x’, only the first term was considered in the Taylor expansion of the fractional part. Therefore the computation is done as follows:

```
log2_fraction_part = (fraction_part) / log2
```

Both the integer and fractional values are summed and used as log2_x quantized values in the training of the CNN model. Using log form reduces the precision of input data and prepares it for low-complexity arithmetic operations in the model's training. And utilizing log_quantized inputs before each layer simplifies the upcoming operations as log can convert the multiplication operations to add operations. The two main advantages of applying this log quantization at multiple stages are:

- Efficient Computation: Logarithmic quantization reduces the cost of high-dimensional operations by replacing them with simpler arithmetic
- Hardware Efficiency: Conversion of multiplication to addition is better for hardware efficiency as they can be performed faster at the hardware level.

The architecture and trainable parameters are depicted in Fig 2.2.

Model: "log_cnn_7"

Layer (type)	Output Shape	Param #
conv2d_32 (Conv2D)	(64, 32, 32, 32)	320
batch_normalization_30 (BatchNormalization)	(64, 32, 32, 32)	128
conv2d_33 (Conv2D)	(64, 32, 32, 32)	9,248
max_pooling2d_21 (MaxPooling2D)	?	0
conv2d_34 (Conv2D)	(64, 16, 16, 64)	18,496
batch_normalization_31 (BatchNormalization)	(64, 16, 16, 64)	256
max_pooling2d_22 (MaxPooling2D)	?	0
conv2d_35 (Conv2D)	(64, 8, 8, 128)	73,856
batch_normalization_32 (BatchNormalization)	(64, 8, 8, 128)	512
max_pooling2d_23 (MaxPooling2D)	?	0
flatten_13 (Flatten)	(64, 2048)	0
dense_15 (Dense)	(64, 256)	524,544
batch_normalization_33 (BatchNormalization)	(64, 256)	1,024
dense_16 (Dense)	(64, 10)	2,570

Total params: 1,890,944 (7.21 MB)
Trainable params: 629,994 (2.40 MB)
Non-trainable params: 960 (3.75 KB)
Optimizer params: 1,259,990 (4.81 MB)

Fig 2.2 Architecture and parameters of the trained CNN model

A final accuracy of 99.23% was achieved with this CNN model. To compare the accuracy and training time of this method, this method was compared with a standard CNN MNIST classifier and simplified Bitwise MNIST classifier. The codes of these models along with their vhdl implementations are given at the end of this document.

4. Results

The implementation results obtained after training and comparing the various CNN models are depicted below. Fig 3.1 depicts the Training, validation accuracy and loss in a simplified Log_Bitwise_MNIST Classifier, Fig 3.2 depicts Training, validation accuracy and loss in a Standard_MNIST_classifier, Fig 3.3 depicts Training, validation accuracy and loss in the proposed implementation and Fig 3.4 depicts the comparison of training times per epoch of all three of the previously mentioned CNN classifiers.

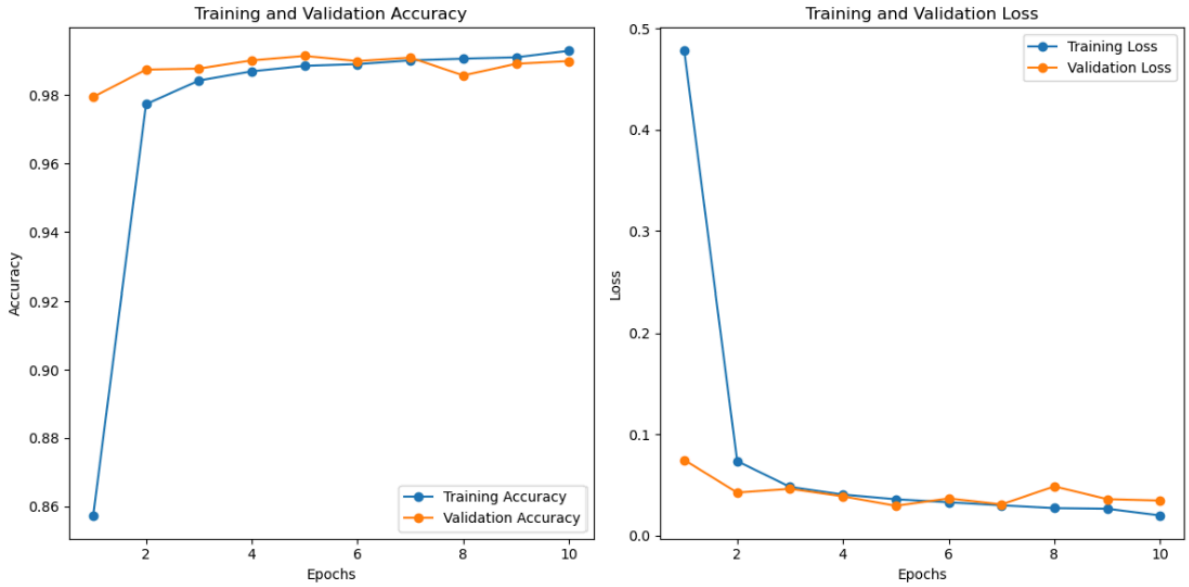

Fig 3.1 Training, validation accuracy and loss in simplified Log_Bitwise_MNIST Classifier

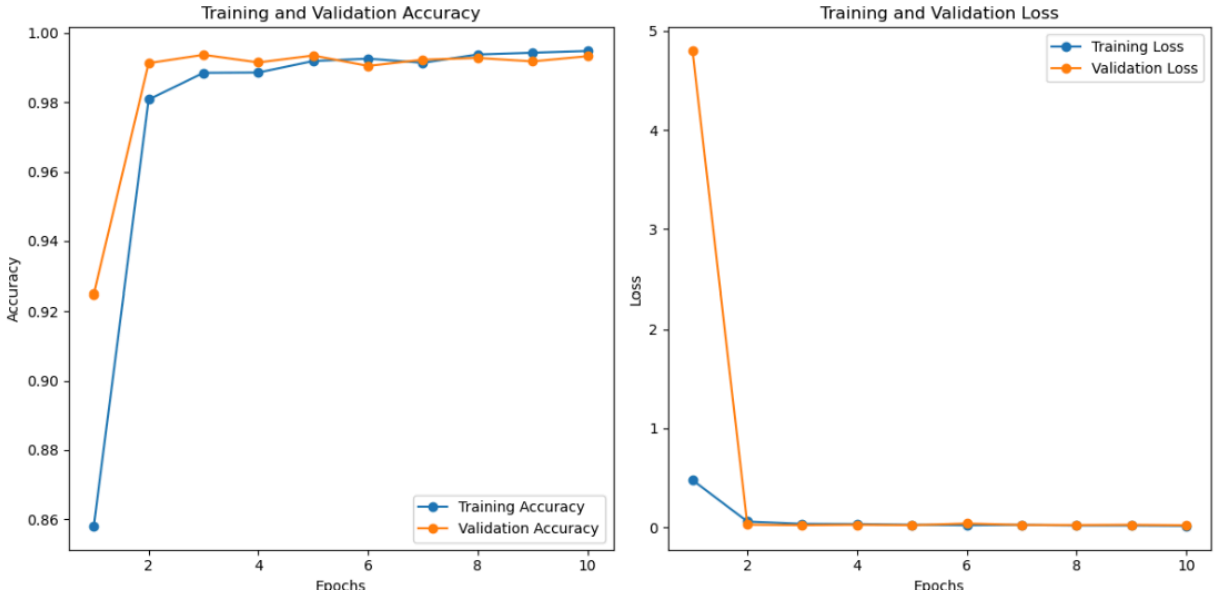

Fig 3.2 Training, validation accuracy and loss in a Standard_MNIST_classifier

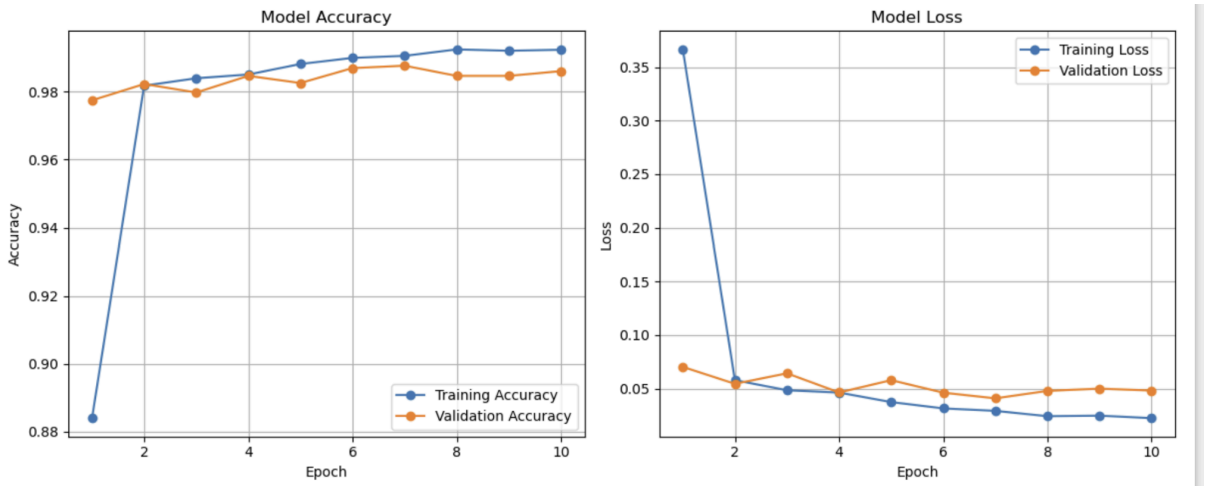

Fig 3.3 Training, validation accuracy and loss in the proposed implementation

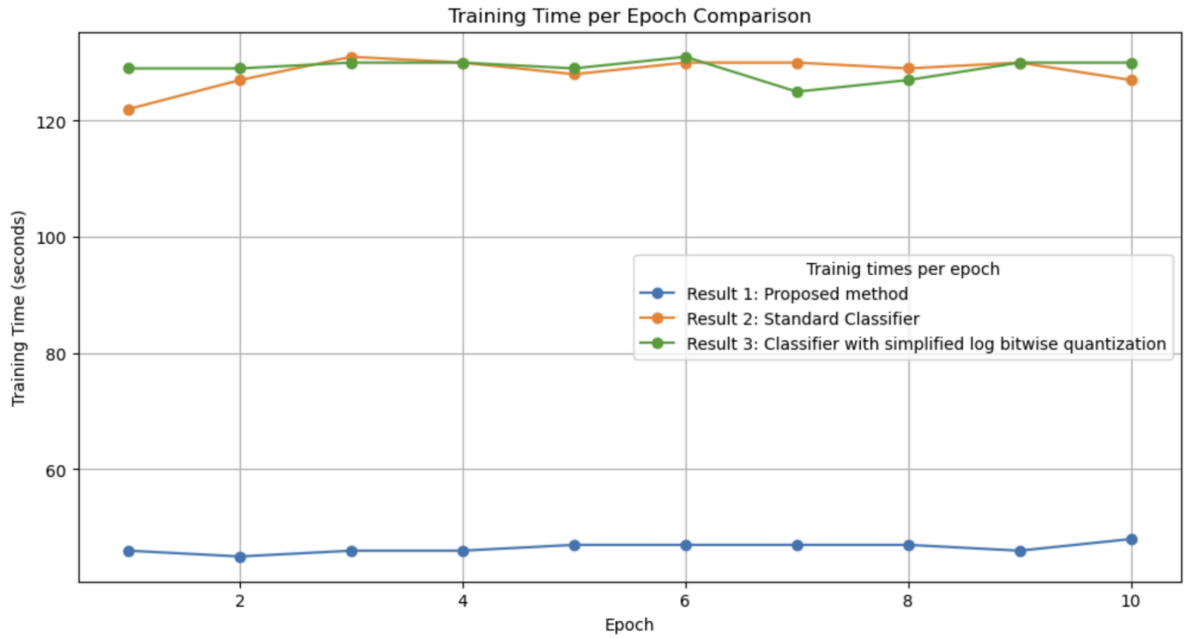

Fig 3.4 Training time per epoch comparison

From Fig 3.1, 3.2 and 3.3 we can clearly infer that the final accuracies obtained by these models are very comparable. Our proposed method achieves a final training accuracy of 99.24% and an accuracy of 98.70% accuracy on the test set. On the other hand, the other two models also give similar accuracies about ~99.4% and test set accuracies of around ~99.3%. Even though the test set accuracies of the other two classifiers are slightly better, the proposed method excels in the training times required per epoch. The log quantization techniques employed simplify the training times where each epoch takes about 60 ms to train which is considerably lower as compared to the other classifiers compared in the paper.

We also attempted to simulate the log_quantization used in this project in vhdl. The simulation results obtained are depicted as follows:

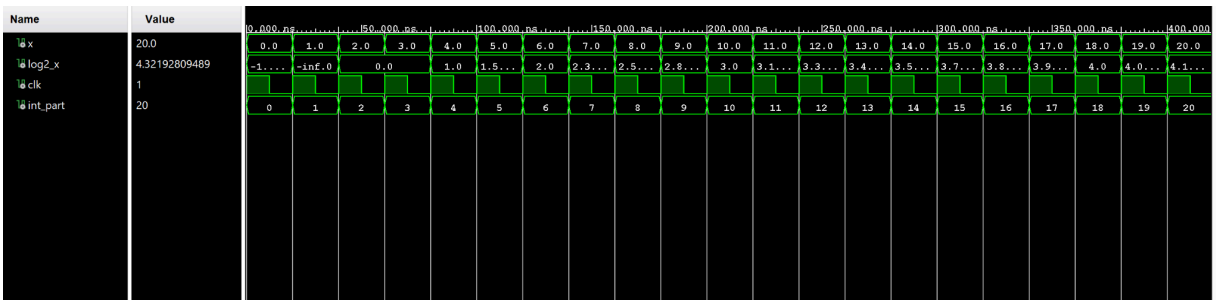

Fig 3.5 VHDL simulation results

5. Conclusion

The method proposed in this project report is hardware efficient as we attempt to simplify the computations in the training process using log quantization and integrating taylor series approximation and lookup tables in log quantization process. However this process is faster in training as compared to a comparable standard model, this is a simplified implementation of a very basic classifier. Further enhancements can be made to improve the overall speed and accuracy in more varied and diverse cases.

Future Work

This work can be further extended in the following ways:

- Using a different dataset with complex and diverse features.
- Using in real world conditions where unpredictability is high.
- Comparing this method with different log bases (log base 2 was used in our implementation).
- Using more taylor series terms and comparing the results.

References

- Zhao, Y., Wang, D., Wang, L., & Liu, P. (2018). A faster algorithm for reducing the computational complexity of convolutional neural networks. Algorithms, 11(10), 159. https://doi.org/10.3390/a11100159
- https://arxiv.org/abs/1506.01195
- Miyashita, D., Lee, E. H., & Murmann, B. (2016). Convolutional neural networks using logarithmic data representation. arXiv preprint arXiv:1603.01025.
- B. F. Cockburn, D. G. Elliott, A. Alimohammad, J. Han (2020), Design of Arithmetic Circuits, Department of Electrical and Computer Engineering, University of Alberta
- Bruce F. Cockburn & Jie Han (2024), Introduction to the VHDL Hardware Description Language, Department of Electrical and Computer Engineering, University of Alberta
- https://math.stackexchange.com/questions/878374/taylor-series-of-ln1x
- O'Shea, K. (2015). An introduction to convolutional neural networks. arXiv preprint arXiv:1511.08458.
- IBM. (n.d.). Convolutional neural networks. IBM. Retrieved December 7, 2024, from https://www.ibm.com/topics/convolutional-neural-networks
- Zoumana Keita. (2023, November 14). An introduction to convolutional neural networks (cnns). DataCamp. https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns
- https://www.tensorflow.org/datasets/keras_example

CNN classifier code:

Proposed method python code:

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

log2 = tf.math.log(2.0)
log2_lookup_np = np.log2(np.arange(1, 257))
log2_lookup = tf.constant(log2_lookup_np, dtype=tf.float32)

def int_frac(x):
    x_clipped = tf.clip_by_value(x, 0.01, 256.0)
    int_part = tf.cast(tf.math.floor(x_clipped), tf.int32)
    fraction_part = x_clipped - tf.cast(int_part, tf.float32)
    return int_part, fraction_part

def taylor_approx(x, n_terms=2):
    x_clipped = tf.clip_by_value(x, 0.01, 256.0)
    int_part, fraction_part = int_frac(x_clipped)
    log2_int_part = tf.gather(log2_lookup, tf.clip_by_value(int_part, 0,
255))
    log2_fraction_part = fraction_part / log2
    return log2_int_part + log2_fraction_part

def log_quantize(x):
    log2_x = taylor_approx(x, n_terms=2)
    return log2_x

class LogCNN(tf.keras.Model):
    def __init__(self, input_shape, num_classes):
        super(LogCNN, self).__init__()
        self.conv1 = layers.Conv2D(32, (3, 3), padding='same',
activation='relu')
        self.bn1 = layers.BatchNormalization()
        self.conv2 = layers.Conv2D(32, (3, 3), padding='same',
activation='relu')
        self.pool1 = layers.MaxPooling2D((2, 2))
        self.conv3 = layers.Conv2D(64, (3, 3), padding='same',
activation='relu')
        self.bn2 = layers.BatchNormalization()
        self.pool2 = layers.MaxPooling2D((2, 2))
        self.conv4 = layers.Conv2D(128, (3, 3), padding='same',
activation='relu')
        self.bn3 = layers.BatchNormalization()
        self.pool3 = layers.MaxPooling2D((2, 2))
        self.flatten = layers.Flatten()
        self.fc1 = layers.Dense(256, activation='relu')
```

```
        self.bn4 = layers.BatchNormalization()
        self.fc2 = layers.Dense(num_classes, activation='softmax')

    def call(self, inputs):
         x = log_quantize(inputs)
         x = self.conv1(x)
        x = log_quantize(x)
        x = self.bn1(x)
        x = self.conv2(x)
        x = log_quantize(x)
        x = self.pool1(x)
        x = self.conv3(x)
        x = log_quantize(x)
        x = self.bn2(x)
        x = self.pool2(x)
        x = self.conv4(x)
        x = log_quantize(x)
        x = self.bn3(x)
        x = self.pool3(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = log_quantize(x)
        x = self.bn4(x)
        x = self.fc2(x)
        return x

if __name__ == "__main__":
    (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()
    x_train = x_train.astype("float32") / 255.0
    x_test = x_test.astype("float32") / 255.0
    x_train = np.expand_dims(x_train, -1)
    x_test = np.expand_dims(x_test, -1)
    x_train = tf.image.resize(x_train, (32, 32))
    x_test = tf.image.resize(x_test, (32, 32))

    y_train = tf.keras.utils.to_categorical(y_train, 10)
    y_test = tf.keras.utils.to_categorical(y_test, 10)

    input_shape = (32, 32, 1)
    num_classes = 10
    model = LogCNN(input_shape, num_classes)


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                  loss="categorical_crossentropy", metrics=["accuracy"])
```

```
    model.fit(x_train, y_train, batch_size=64, epochs=10,
validation_split=0.2)

    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
    print(f"Test Accuracy: {test_acc:.4f}")
```

Standard MNIST classifier python code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape((x_train.shape[0], 28, 28,
1)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1)).astype('float32')
/ 255

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

model = models.Sequential()

model.add(layers.Conv2D(64, (3, 3), padding='same', activation='relu',
input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))
```

```
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, batch_size=128,
validation_split=0.1)

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test Accuracy: {test_acc:.4f}")
```

VHDL codes:

Integer and Fraction value generation:

```
----------------------------------------------------------------------------
----------
-- Company:
-- Engineer:
--
-- Create Date: 12/17/2024 05:16:31 AM
-- Design Name:
-- Module Name: int_frac - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
----------
library ieee;
use ieee.std_logic_1164.all;
```

```
entity int_frac is
    port (
        x : in REAL;
        int_part : out INTEGER;
        frac_part : out REAL
    );
end entity int_frac;

architecture behavioral of int_frac is
begin
    process(x)
        variable int_result : INTEGER;
        variable frac_result : REAL;
    begin
        int_result := INTEGER(x);
        frac_result := x - REAL(int_result);

        int_part <= int_result;
        frac_part <= frac_result;
    end process;
end architecture behavioral;
```

Log2 Lookup table:

```
----------------------------------------------------------------------------
---------
-- Company:
-- Engineer:
--
-- Create Date: 12/17/2024 05:15:58 AM
-- Design Name:
-- Module Name: log2_lookup - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
---------
```

```
library ieee;
use ieee.std_logic_1164.all;

entity log2_lookup is
    port (
        int_part : in INTEGER;
        log2_out : out REAL
    );
end entity log2_lookup;

architecture behavioral of log2_lookup is
    type lut_array is array (0 to 255) of REAL;
    constant log2_lut : lut_array := (
        0.0,             -- log2(0)
        0.0,             -- log2(1) = 0
        1.0,             -- log2(2)
        1.58496250072,   -- log2(3)
        2.0,             -- log2(4)
        2.32192809489,   -- log2(5)
        2.58496250072,   -- log2(6)
        2.80735492206,   -- log2(7)
        3.0,             -- log2(8)
        3.16992500144,   -- log2(9)
        3.32192809489,   -- log2(10)
        3.45943161864,   -- log2(11)
        3.58496250072,   -- log2(12)
        3.70043971814,   -- log2(13)
        3.80735492206,   -- log2(14)
        3.90689059561,   -- log2(15)
        4.0,             -- log2(16)
        4.08746284125,   -- log2(17)
        4.16992500144,   -- log2(18)
        4.24792751344,   -- log2(19)
        4.32192809489,   -- log2(20)
        -- THIS LOOKUP NEEDS TO  BE FILLED WITH ENTRIES TILL LOG 256, FOR
SIMPLICITY VALUES TILL LOG 20 ARE INCLUDED
        others => 0.0
    );

begin
    process(int_part)
    begin
        if int_part >= 0 and int_part < log2_lut'length then
            log2_out <= log2_lut(int_part);
        else
```

```
            log2_out <= 0.0;
        end if;
    end process;
end architecture behavioral;
```

Taylor Approximation:

```
----------------------------------------------------------------------------
----------
-- Company:
-- Engineer:
--
-- Create Date: 12/17/2024 05:17:20 AM
-- Design Name:
-- Module Name: taylor_approx - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
----------

library ieee;
use ieee.std_logic_1164.all;

entity taylor_approx is
    port (
        frac_part : in REAL;
        log2_frac_out : out REAL
    );
end entity taylor_approx;

architecture behavioral of taylor_approx is
    constant log2_inv : REAL := 1.442;
```

```
begin
    process(frac_part)
        variable scaled_result : REAL;
    begin
        scaled_result := frac_part * log2_inv;
        log2_frac_out <= scaled_result;
    end process;
end architecture behavioral;
```

Log Quantize file(Top level entity):

```
----------------------------------------------------------------------------
----------
-- Company:
-- Engineer:
--
-- Create Date: 12/17/2024 05:17:55 AM
-- Design Name:
-- Module Name: log_quantize - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
----------
library ieee;
use ieee.std_logic_1164.all;

entity log_quantize is
    port (
        x : in REAL;
        log2_x : out REAL := 0.0;
        clk : in std_logic
    );
end entity log_quantize;

architecture behavioral of log_quantize is
```

```
    -- Intermediate signals
    signal int_part : Integer;
    signal frac_part : REAL;
    signal log2_int_part : REAL;
    signal log2_frac_part : REAL;
    signal log2_sum : REAL;
begin
    -- Instantiate the components
    part_1: entity work.int_frac port map(x => x, int_part => int_part,
frac_part => frac_part);
    part_2: entity work.log2_lookup port map(int_part => int_part,
log2_out => log2_int_part);
    part_3: entity work.taylor_approx port map(frac_part => frac_part,
log2_frac_out => log2_frac_part);

    process(clk)
    begin
        if rising_edge(clk) then
            log2_sum <= log2_int_part + log2_frac_part;
            log2_x <= log2_sum;
        end if;
    end process;
end architecture behavioral;
```

Testbench file:

```
----------------------------------------------------------------------------
----------
-- Company:
-- Engineer:
--
-- Create Date: 12/17/2024 05:24:23 AM
-- Design Name:
-- Module Name: tb_log_quantize - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
```

```
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
----------
library ieee;
use ieee.std_logic_1164.all;

entity tb_log_quantize is
end entity tb_log_quantize;

architecture behavioral of tb_log_quantize is
    component log_quantize
        port (
            x : in REAL;
            log2_x : out REAL;
            clk : in std_logic
        );
    end component;

    signal x : REAL;
    signal log2_x : REAL;
    signal clk : std_logic := '0';

begin
    clock_process : process
    begin
        clk <= NOT clk;
        wait for 10 ns;
    end process;

    UUT: log_quantize
        port map (x => x, log2_x => log2_x, clk => clk);

    stimulus: process
    begin
        x <= 0.0;
        wait for 20 ns;

        x <= 1.0;
        wait for 20 ns;

        x <= 2.0;
        wait for 20 ns;

        x <= 3.0;
```

```
        wait for 20 ns;

        x <= 4.0;
        wait for 20 ns;

        x <= 5.0;
        wait for 20 ns;

        x <= 6.0;
        wait for 20 ns;

        x <= 7.0;
        wait for 20 ns;

        x <= 8.0;
        wait for 20 ns;

        x <= 9.0;
        wait for 20 ns;

        x <= 10.0;
        wait for 20 ns;

        x <= 11.0;
        wait for 20 ns;

        x <= 12.0;
        wait for 20 ns;

        x <= 13.0;
        wait for 20 ns;

        x <= 14.0;
        wait for 20 ns;

        x <= 15.0;
        wait for 20 ns;

        x <= 16.0;
        wait for 20 ns;

        x <= 17.0;
        wait for 20 ns;

        x <= 18.0;
        wait for 20 ns;
```

```
        x <= 19.0;
        wait for 20 ns;

        x <= 20.0;
        wait for 20 ns;

        wait;
    end process;
end architecture behavioral;
```