Project 2 Report
Artificial Intelligence on Chip, Winter 2020

Hao Jen Chien UID: 405219534
Ananya Ravikumar UID: 205431526

***Abstract*—In this project, we perform 8-bit, fixed point quantization on the parameters of SqueezeNet v1.1, and compare the inference accuracy of the models before and after quantization.**

I. SqueezeNet Architecture

SqueezeNet [1] is a highly compressed convolutional neural network which is able to provide Alexnet-level [2] accuracy with 50x fewer parameters. SqueezeNet achieves this size reduction by:

- Replacing a majority 3x3 filters (9 parameters) with 1x1 filters (1 parameter).
- Reducing the number of input channels to 3x3 filters. This reduces parameters because the number of parameters in a convolutional layer with 3x3 filter is given by (3x3)*(no. of input channels)*(no. of filters).
- Down-sampling at a later stages in the network to increase accuracy.
- The design a special unit called a *fire module*, which consists of a *squeeze layer* containing only 1x1 filters, and an *expand layer* containing both 1x1 and 3x3 filters. The number of 1x1 and 3x3 filters in these layers of the module are hyperparameters. The architecture of squeezenet is shown in figure 1.

The pre-trained model used in this project is SqueezeNet V1.1. It has 2.4x less computation and and slightly fewer parameters than the original SqueezeNet (SqueezeNet V1.0) without any change in accuracy [3].

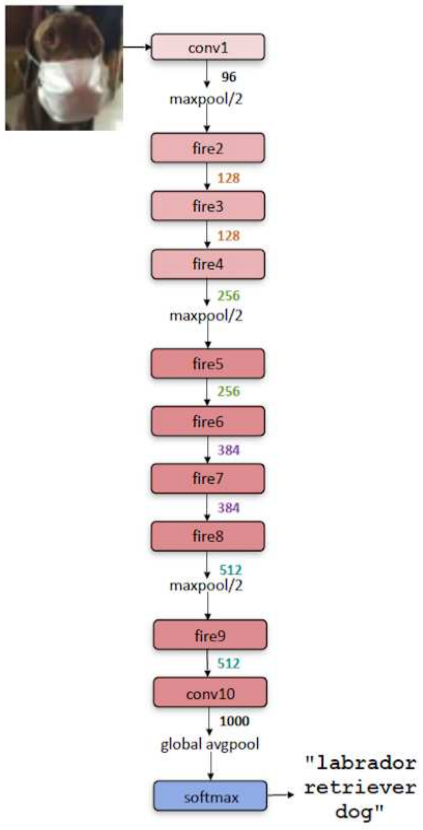

Fig. 1. SqueezeNet Structure[1]

II. Fixed Point Representation

Fixed point representation of binary numbers assumes that there are a fixed number of bits before and after the binary point. For a word length of wl bits with fl bits after the binary point, there are $wl - fl$ bits before the binary point. If the bits are numbered as 0 to $wl - 1$ from the LSB to the MSB, the value of the number is given by:

$$val = \sum_{i=0}^{wl-2} b_i \times 2^{i-fl} \qquad \text{if } b_{wl-1} = 0 \tag{1}$$

$$val = -2^{wl-1} + \sum_{i=0}^{wl-2} b_i \times 2^{i-fl} \qquad \text{if } b_{wl-1} = 1 \tag{2}$$

The most significant bit (MSB) is the sign-bit. An example of fixed-point representation is shown below. $(01001.111)_2$, an 8-bit fixed point number with $fl = 3$, has the value

$$\begin{aligned}(01001.111)_2 &= 1 \times (2^{-3} + 2^{-2} + 2^{-1} + 2^0 + 2^3) && (3)\\ &= 0.125 + 0.25 + 0.5 + 1 + 8 && (4)\\ &= 9.875 && (5)\end{aligned}$$

III. Quantization

Quantization refers to the idea of restricting the range of values that a continuous quantity can take to a specific set of values. Quantization of neural network parameters is done to:

1) reduce the model size by reducing parameter sizes
2) reduce computation time
3) make the implementation of the model hardware-friendly.

IV. Implementation

Below is the steps of implementation of this project:

- This project was implemented in Python 3.6.9 using the *tensorflow* library [4].
- A pre-trained model of SqueezeNet v1.1 trained on ImageNet [5] was downloaded from [6].
- The model was quantized as shown in the next section.
- The tvm library was used to increase the speed and efficiency of computation.
- Inference was done on the original and quantized models using Google Colaboratory.
- The validation data for the ImageNet dataset consists of 50,000 color images which belong to one of 1000 classes.

V. Method

A. 8-Bit Fixed Point Quantization

Two different methods of Quantization were attempted:

Rounding based method: In order to lose as little information as possible, we round each parameter to a permissible value (one that can be represented with 8 bits) that is closest to its original value. For values larger than the maximum value that can be stored in 8-bit fixed point, they will be stored as the maximum value. The concept is applied to values smaller than the minimum that can be stored in 8-bit fixed point.

1) Find max value (max_v) that can be represented with (wl,fl)
2) Find sign of each value to be quantized.
3) Round each value to the smallest integer closest to it.
4) Find the fractional part of the result by converting the fractional part to binary upto ‘fl’ bits.
5) Add the integer part to the fractional part with the appropriate signs.
6) Set the LSB to 1 if the error due to the bits after fl is greater than 2^{-fl}.
7) Use tf.clip_by_value(val_fp,-max_v,max_v)to fit the value in the permissible range.

Offset based method: Use the formula:

$$val_{quant} = val_{min} + step.round(\frac{val - val_{min}}{step}) \quad (6)$$

to calculate the quantized value val_{quant} from the original data - $value$. val_{min} is the minimum value in the set of parameters being quantized.

1) Calculate integer and fractional parts as shown in Method 1 for $(val - val\_min)$ instead of just val, and add them to get val_fp.
2) Use tf.clip_by_value(val_fp,-max_v,max_v)to fit the value in the permissible range.
3) Return min_val + the result of the previous step as the final parameter value.

B. Search Strategy for optimal fraction length

The choice of fraction length, fl, is based on the tradeoff between magnitude and precision. A larger integer part allows larger numbers to be represented, while a larger number of bits after the binary points allows higher accuracy. The largest integer that can be represented using fl bits after the binary point is $\pm 2^{wl-fl-1}$, while the most precision that can be obtained is 2^{-fl}.

An iterative search was performed to find the optimal length fl of the fractional part of the parameters by calculating the mean squared error. fl was initialised with the value 1, and was increased by 1 in each iteration up to $wl - 2$. The mean squared error between the original parameters and the quantized parameter was computed in each iteration, the value of fl which minimised this error was chosen.

C. Evaluation

The performance of the quantized model was evaluated on the validation set of ImageNet (50,000). This was done on the Google Colab platform.

1) Pre-processing Images: All images from the ImageNet validation dataset were pre-processed before they were fed to the network. The images had a shape (3 x H x W), where H and W are expected to be at least 224. The transform function in Pytorch was used to covert images to tensorflow tensor with a range of [0, 1], normalized mean = [0.485, 0.456, 0.406], and standard deviation = [0.229, 0.224, 0.225] for each channel [3].

2) Getting Accuracy: The images were then input into the network with batch size of 100. The output was a distribution of probabilities that an image belongs to a class. After finding the class with the highest probability predicted by the network and comparing to the correct class, average accuracy of each batch was calculated. The final average accuracy was obtained by computing the total number of correctly classified images across all batches and dividing it by the total number of images (50000).

VI. Results

Model Type	Top 1 Accuracy (%)	Top 5 Accuracy (%)
Original	58.18	80.57
Quantized v1	35.31	64.14
Quantized v2	51.44	74.46

TABLE I
Inference Accuracy

Table I shows the Top-1 and Top-5 inference accuracy of the quantized network and full-precision network on the validation set of ImageNet of 50,000 images. Note that the network was not re-trained after quantization. SqueezeNet v1_1 has 1,235,496 parameters and has a size of 5MB [7]. After quantization, the network as 1,231,552 parameters, and each of these 32-bit parameters can be stored in 8-bits. Thus, the model size can be reduced by 3.7MB to around 1.3MB. In case of Method-2, some additional space will be required

Layers	wl (feat. map, conv. layer, bias)	fl(feat. map, conv. layer, bias)
1	8,8,16	4,6,14
2	8,8,16	3,6,15
3	8,8,16	3,7,15
4	8,8,16	3,7,14
5	8,8,16	1,6,13
6-25	8,8,16	1,7,15

TABLE II
WL,FL VALUES FOR METHOD 2

to store the 64-bit minimum value (offset) for each set of quantized values. We can see that with storing extra minimum values, the quantized model was able to achieve a much better result of accuracy, while the quantized model without minimum value performed poorly, having a 23% reduction of accuracy for Top 1 and 15% reduction of accuracy for Top 5.

Table II shows the floating length fl values selected by each layer. Throughout the whole model, same type of layers has a similar floating length fl. It is interesting to note that except for the first few layers, the feature map has a very low fl value, while the convolution layers and biases have high fl values. We can see that inner convolution layers have a longer floating length and outer convolution layers have shorter floating length. Bias layers are set to have 16-bit fixed point with very large floating length fl values. The possible explanation is that the values of biases are very small numbers less than 1 so that they need long floating points to preserve the best values.

VII. CONCLUSIONS

In this project, we have implemented 8-bit fixed point quantization on SqueezeNet by using a a) rounding based approach and b) an offset based approach. The first approach yields a quantized model with a test accuracy of $\tilde{3}5.3\%$,which is significantly lower than that of the original model. The second approach leads to a test accuracy of $\tilde{5}1.44\%$, which is about 7% lower than the original mode. The creators of SqueezeNet have shown that Deep Compression (a combination of pruning, trained quantization and Huffman coding) [8] can be applied to SqueezeNet without causing any loss in accuracy. However, the network is re-trained after compression in their implementation.

In our project, the mean square errors have a range of 0.01 to 1 for quantized weights in each layer. We believe that simple rounding causes errors to accumulate in the network and leads to accuracy loss. We believe that the offset based method works well in this situation becuase it is able to reduce the error between the original and quantized parameters by representing a larger range of numbers. Re-training the network, trained selection of weights achieved by k-means clustering and greater control over the choice of optimal fl could be some ways to improve accuracy.

Link to code (notebook):
https://colab.research.google.com/drive/1le5o3wJJsDfATqJMxCiOT0bygN5pHasH

REFERENCES

[1] F. Iandola, M. Moskewicz, K. Ashraf, S. Han, W. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and textless1mb model size,” 02 2016.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’12, (Red Hook, NY, USA), p. 1097–1105, Curran Associates Inc., 2012.

[3] P. Team, “Squeezenet— pytorch.” https://pytorch.org/hub/pytorch_vision_squeezenet/, 2018.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.

[6] “Torchvision.models.” https://pytorch.org/docs/stable/torchvision/models.html, 2018.

[7] “Squeezenet v1.1: Wolfram neural net repository.” https://resources.wolframcloud.com/NeuralNetRepository/resources/SqueezeNet-V1.1-Trained-on-ImageNet-Competition-Data, 2018.

[8] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2015.

```
import numpy as np
import tensorflow as tf
import math
'''
utility function

with with block.suppress_stdout_stderr():
    your code

To hide stdout/stderr output i.e. from Tensorflow initialzation
'''
from . import suppress_stdout_stderr as block

def tf_symbolic_convert(value, wl, fl):
    '''
    version -1

    Convert float numpy.array/tf.Tensor to wl-bit low precision data
with Tensorflow API

    Inputs：
    - value : input data in numpy.array/tf.Tensor format
    - wl : word length of the data format to convert
    - fl : fraction length (exponent length for floating-point)

    Returns:
    - val_fp : tf.Tensor as the symbolic expression for quantization
    '''
    # ==================================================================
#
    # YOUR CODE HERE:
    #   tf.clip_by_value could be helpful
    # ==================================================================
#
    max_v = 0
    for i in range(int(wl-1),0,-1):
      max_v +=2**(i-fl-1)

    if type(value).__module__ == np.__name__:
       value = tf.convert_to_tensor(value)

    value_sign = tf.sign(value)
    val_min = tf.reduce_min(value)
    val_fp = tf.abs(value-val_min)
    val_fp = tf.floor(val_fp)*values_sign

    values_frac = tf.abs(value-val_min)-tf.abs(val_fp)

    for i in range(fl):
      binary_base = 2 **-(i+1)
      values_frac = values_frac*2
      select = tf.greater_equal(values_frac, 1)
      select = tf.cast(select, value.dtype)
      values_frac = values_frac - select
      val_fp = val_fp + select * binary_base * values_sign
    remaining = tf.greater_equal(tf.abs(value-val_fp), 2**-(fl+1))
    remaining = tf.cast(remaining, value.dtype)

    val_fp = val_fp + remaining * 2**-fl * value_sign
    val_fp = clip_by_value(val_fp, -max_v, max_v)

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
    return val_fp

def tf_symbolic_convert(value, wl, fl):
    '''
    #version 2
    Convert float numpy.array/tf.Tensor to wl-bit low precision data
with Tensorflow API

    Inputs：
    - value : input data in numpy.array/tf.Tensor format
    - wl : word length of the data format to convert
    - fl : fraction length (exponent length for floating-point)

    Returns:
    - val_fp : tf.Tensor as the symbolic expression for quantization
    '''
    # ================================================================
#
    # YOUR CODE HERE:
    #   tf.clip_by_value could be helpful
    # ================================================================
#
    max_v = 0
        for i in range(int(wl-1),0,-1):
          max_v +=2**(i-fl-1)

        if type(value).__module__ == np.__name__:
                value = tf.convert_to_tensor(value)

        val_min = tf.reduce_min(value)

        val_fp = tf.abs(value-val_min)
        value_sign = tf.sign(value-val_min)

        val_fp = tf.floor(val_fp)*value_sign

        values_frac = tf.abs(value-val_min)-tf.abs(val_fp)

        for i in range(fl):
          binary_base = 2 **-(i+1)
          values_frac = values_frac*2
          select = tf.greater_equal(values_frac, 1)
          select = tf.cast(select, value.dtype)
          values_frac = values_frac - select
          val_fp = val_fp + select * binary_base * value_sign
        remaining = tf.greater_equal(tf.abs(value-val_fp), 2**-(fl+1))
        remaining = tf.cast(remaining, value.dtype)

        val_fp = val_fp + remaining * 2**-fl * value_sign
        val_fp = tf.clip_by_value(val_fp, -max_v, max_v) + val_min
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
    return val_fp


class Qnn:
    def __init__(self):
        pass

    # dtype convertion: basic functions
    def to_fixedpoint(self, data_i, word_len, frac_len):
        return tf_symbolic_convert(data_i, word_len, frac_len)

    # utility function to convert symbolically or numerically
    def convert(self, data_i, word_len, frac_len, symbolic=False):
        if symbolic is True:
            data_q = self.to_fixedpoint(data_i, word_len, frac_len)
        else:
            with tf.Graph().as_default():
                data_q = self.to_fixedpoint(data_i, word_len,
frac_len)
                with block.suppress_stdout_stderr():
                    with tf.Session() as sess:
                        data_q = sess.run(data_q)
        return data_q

    # error measurement
    def difference(self, data_q, data_origin):
        '''
        Compute the difference before and after quantization

        Inputs：
        - data_q: a numpy array of quantized data
        - data_origin: a numpy array of original data

        Returns:
        - dif : numerical value of quantization error
        '''
        #
============================================================ #
        # YOUR CODE HERE:
        #   implement mean squared error between data_q and
data_origin
        #
============================================================ #

        dif = np.sum((data_q - data_origin)**2)

        #
```

```
================================================================== #
        # END YOUR CODE HERE
        #
================================================================== #
        return dif

    # search policy
    def search(self, data_i, word_len):
        '''
        Search for the optimal fraction length that leads to minimal
quantization error for data_i

        Inputs :
        - data_i : a numpy array of original data
        - word_len : word length of quantized data

        Returns:
        - fl_opt : fraction length (python built-in int data type)
that leads to minimal quantization error
        '''
        #
================================================================== #
        # YOUR CODE HERE:
        #   compute quantization error for different fraction lengths
        #   and determine fl_opt
        #
================================================================== #
        error = []
        frac_len = range(word_len)

        for f in frac_len:
          data = self.convert(data_i, word_len, f)
          error.append(self.difference(data, data_i))

        max_index = error.index(min(error))
        fl_opt = frac_len[max_index]

        #
================================================================== #
        # END YOUR CODE HERE
        #
================================================================== #
        return fl_opt

    # granularity
    def apply(self, data_i, word_len):
        fl_opt = self.search(data_i, word_len)
        data_q = self.convert(data_i, word_len, fl_opt)
        return data_q, fl_opt
```