March 24th, 2020

Project 2 - YoloV3 Network Quantization
ECE209 AI On Chip Winter 2020

Bryan Bednarski 005428092
Matt Nicholas 305431314

Introduction

The purpose of this project is to demonstrate the accuracy trade-offs in neural network image classification tasks when quantization is performed on the weights, biases, and feature maps of the Yolo-V3 object detection network. In this project, we aim to demonstrate the impact of quantization by representing values for these features as 8-bit and 16-bit floating point values rather than their default 32-bit floating point representation. In this implementation, we make this conversion symbolically, so there will be no overall impact on the total size of the trained network from the quantization. However, it will allow for the analysis of the classification and detection accuracy that would arise from implementing actual quantization. Additionally, at the end of the report we compress the quantized networks, and compare the ability of these mainstream compression algorithms to reduce the size of symbolically quantized networks.

Making these changes to the network using the TVM compiler requires a number of steps. The aim of this report is to guide the reader through the steps taken to achieve successful quantization, and discuss the results and trade-offs of using a quantized model. The preliminary steps in this process involve compiling the network with the TVM compiler so that the high level quantization scripts can be run for this exercise. Next, we pre-process a single image as a reference for the TVM compiler so that the output of a quantization procedure can produce immediate evaluation results. The compiler generates .pb files for the non-quantized and quantized versions of the Yolo-V3 network. We then load this network back into the original Yolo-V3 framework for final evaluation. For the Yolo-V3 framework, this evaluation involves both classifying recognized objects within images but additionally placing labeled bounding boxed around these objects and showing a new image with these bounding boxes.

Ultimately, the Yolo-V3 network is different from a number of the other networks being analyzed in this class because the network's task is to perform object detection within images, rather than image classification. As a result all functions are self contained, including the image pre-processing and the evaluation of the network and must be re-implemented from the Yolo-V3 source code for this project. Additionally, the evaluation criteria for this network uses a mean average precision (mAP) with a minimum 0.5 intersection over union evaluation. This quantification criteria will be used and discussed extensively throughout the report.[1]

1. Image Pre-Processing

After compiling our network with TVM successfully, we pre-process a reference input image in order to be compatible with the network. This both shows us immediate accuracy results, and also verified that the correct image input is being considered. This is done by running the main.py script in the Quantization_PJ2 repository.

Image pre-processing is done as follows: The image shown below in Figure 1 begins as a 333x555x3 .jpeg image, that we must convert to a .npy file with dimensions 1x416x416x3 for the Yolo-V3 network (loaded from .onnx) to accept as valid input.

Figure 1: Original reference image to convert to .npy

Performing this pre-processing requires just two prominent steps that were described in the Yolo framework. Before either of these steps the image must be converted loaded as PIL image, and converted to a tensor. Next, it can be zero-padded to a square 555x555x3 image, then resized to 416x416x3. Once resized, we can convert the tensor to a numpy array and save the file as a .npy. See Figure 2 below, for a flowchart of these operations.

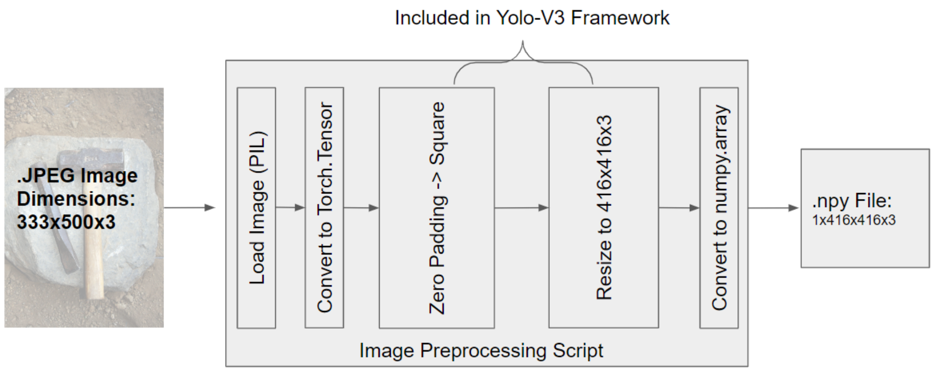


Figure 2: Image pre-processing pipeline, similar to Yolo-V3 original

2. Quantization

2.1. Floating Point Representation

In this project, the TVM compiler represents weights, biases and feature maps as 32-bit floating point values. Following the IEEE 754 standard, a 32-bit single-precision floating point value, can represent a wider range of values than a 32-bit fixed point counterpart with the cost of accuracy at larger values. This is done with the use a a floating point radix, which represents an exponential value rather than a fixed fraction length as would a fixed point value. For example, a signed 32-bit integer can represent values from $2^{31} - 1 = 2,147,483,647$, while an IEEE 754 32-bit float can represent a max value of $(2 - 2^{-23}) * 2^{127} = 3.4028235 * 10^{38}$ [2]. In effect, this exponential bit designation requires values to follow a notation similar to scientific notation for base-2 representations. Only one bit is ever required to designate the sign of the number, positive or negative. Figure 3 below, shows a representation of how a single floating point value's bits can be allocated for a system.

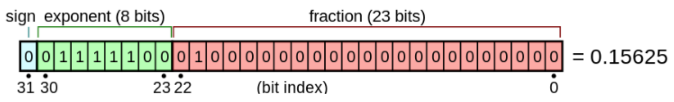


Figure 3: Potential division of bits between mantissa, exponential, and sign for IEEE 754 32-bit float representations.

A potential 8-bit IEEE 754 mini-float representation is shown below in 4. The same rules apply to 8-bit representations as 32-bit representations, but the range of possible representations changes with the number of bits allocated to the mantissa/fraction and exponent. Throughout this project, we will be implementing 8 and 16 bit representations of this standard. It is important to note that in order to represent smaller values with improved precision, a bias is added to whatever value is represented by the exponent bits. This bias is equal to half of the max representation of that number of bits. Therefore, the exponent in scientific representation of the value is negative when the exponent is less than the bias, allowing fractional numbers. The outcome of this design choice is that the user is granted great precision with low values. Figure 5 shows the increased density of values that can be represented for lower-magnitude numbers.

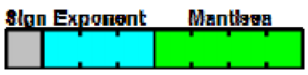


Figure 4: Potential division of bits between mantissa, exponential, and sign for IEEE 754 8-bit float representations[3]

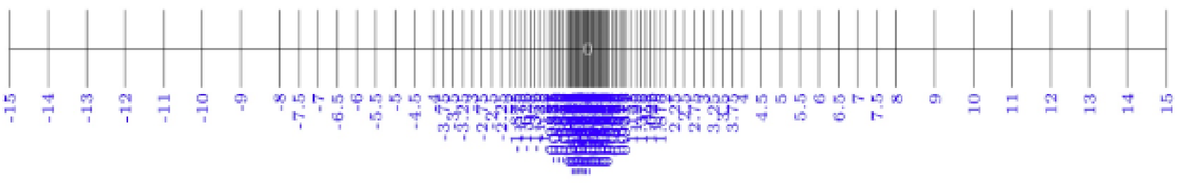


Figure 5: Number line of IEEE 754 8-bit float values, showing improved near-zero fractional precision for floating point values[4]

2.2. Quantization Functions

For this assignment, we were asked to fill in the missing functions from **quantization_utils.py**, so that when quantization was run using the **main.py** script, quantization would be performed properly. The following functions were completed in order to do so:

- **'search(self, data_i, word_len)'** - Search for the optimal fraction length that leads to minimal quantization error for data_i
- **'difference(self, data_q, data_origin)'** - Compute the difference before and after quantization using mean squared error between the values represented in the original graph, and the graph generated through 'tf_symbolic_convert'.
- **'tf_symbolic_convert(value, wl, fl)'** - Convert float numpy array to wl-bit low precision data with Tensorflow API

Quantization is performed by passing a Tensorflow graph, represented by weights, biases, and feature map values to the 'tf_symbolic_convert' function, and replacing the representing graph in the network with the graph that is returned.

The **tf_symbolic_convert function** was completed initially using a built-in Tensorflow API quantization function for 8-bits in order to get a quantized .pb file generated upon which inference could be performed. The built-in function that was used is called: **tf.quantization.fake_quant_with_min_max_args()**, which does a symbolic quantizationf to floating point values of a designated work length. Using this model, we were able to generate a valid quantized .pb file for inference that we could compare our final model to. However, this function does not take a length for the exponent or fractional/mantissa bits and there is no way of knowing how the function is representing these values.

In an attempt to quantize our weights, we initially wrote a quantization function that loops through each value, determines the best exponent to use, and then determines the proper mantissa. The length of the exponent and mantissa is designated by the inputs to the function. We wrote this because the initial assignment specification said that the input is a numpy array. However, it became apparent that it was intended for us to operate on entire tensors, rather than determine the quantization value for individual entries one at a time. Nonetheless, this function reliably translates a 32 bit floating point value into the closest possible value specified by an 8 bit floating point with given exponent and mantissa value. It is not in use now, and takes a very long time to run as it does not perform operations in parallel.

Rather than using an iterative method across numpy arrayed, our final version of the quantization function will instead use the Tensorflow API to process the tensors in parallel using built-in functions. This model will be evaluated the same way and compared to previous quantization schemes.

2.3. Model Quantization

The overall process of quantization is based on generating .pb files, which is a standard for storing the weights, biases, and feature maps of neural networks, to be saved and loaded at a later time without retraining. This file can be loaded as a pretrained network and evaluated at a later time. In our workflow, the TVM compiler first loads the neural network's weights, biases and feature maps from a '.onnx' file to a tensorflow graph. This graph can then be passed through the quantization functions that have been described, to generate a '.pb' file that can be evaluated for its accuracy. The following algorithms define this functionality through the **search**, **difference** and **tf_symbolic_convert** functions

Algorithm 1 below describes the functionality that was implemented in 'tf_symbolic_convert':

Algorithm 1: 'tf_symbolic_convert'

```
input - original_values
bitwise_combinations = $2^{mantissa\_length}$
sign_mask = tf.sign(input_tensor)
exponent_bits_value = tf.math.floor(tf.math.log(input_tensor)/tf.math.log(2))
possible_representations.append(value(exponent_bits_value + 1))
for i in bitwise_combinations do
    next_value = exponent_bits_value * i
    possible_representations - append(next_value)
end for
for i in possible_representations do
    least_difference_representation = index(min(DIFFERENCE(possible_representations(i), original_values))))
end for
return clip_min_max(least_difference_representation)
```

When quantizing a graph the 'tf_symbolic_convert' function must already know the optimal number of bits with which to represent the exponents in IEEE 754 minifloat standard. Algorithm 2 below, shows the iterative process of determining the optimal exponent length for each graph/layer passed to the algorithm.

Algorithm 2: 'search'

```
1: next_graph = a weight, bias or feature map layer
2: min_exponent = 1
3: max_exponent = 6 for 8-bit float, 7 for 16 bit float
4: difference_list = []
5: 'SEARCH' function called:
6: for i in range(max_exponent - min_exponent) do
7:     symbolic_graph = 'TF_SYMBOLIC_CONVERT'(next_graph)
8:     difference_list.append(DIFFERENCE(symbolic_graph, next_graph))
9: end for
10: optimal_exponent = index(min(difference_list))
```

Algorithm 3: 'difference'

```
1: values_quant = quantized_graph.asarray()
2: values_orig = orignal_graph.asarray()
3: return $\sum (values\_quant - values\_orig)^2 / len(values\_quant||values\_orig)$
```

3. Evaluation

3.1. Evaluation Criteria

The Yolo-v3 network was designed to label multiple objects within images, and place bounding boxes around those objects. The output of this network is different from the other networks used for this assignment that simply output and array of likelihoods that the image contains a certain object. Many other other networks are simply classification algorithms, while the Yolo network is a real time object recognition network that can label and bound multiple objects in a a single image. As a result, the evaluation criteria for Yolo-V3 is quantified as a mean average precision (mAP) with a minimum intersection over union of 0.5. This means that in order for a classification to be considered correct, the object must not only be labeled correctly, but also bounded, where at least 50% of the area of the bounding box is within the labeled bounding region. Figure 6 shows an example of this classification criteria with bounding a stop sign.

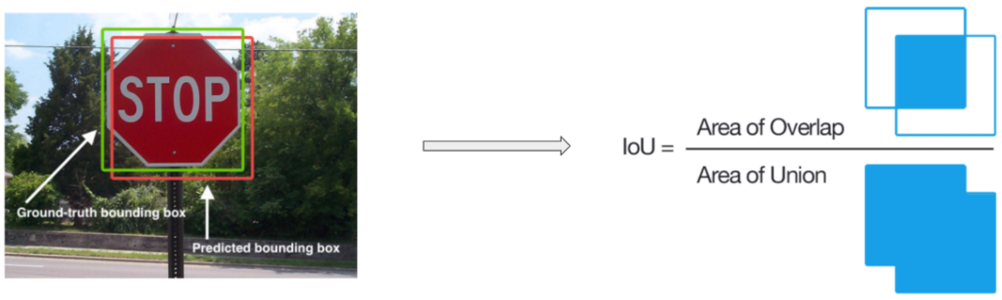
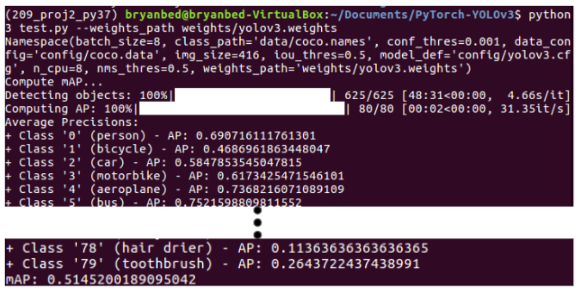


Figure 6: Bounding box overlap and IOU for a stop sign classification.

As a result of its evaluation criteria Yolo-V3 network must be trained on a dataset that both labels and bounds classifiers within images. The **coco** dataset, provided with the Yolo-V3 repository does just that. Therefore, when evaluating the

Yolo-V3 network the primary result over a set of data can be a maximum a posteriori estimate (mAP) reporting a point estimate for the classifiers in the image. Across the entire data set, this mAP average is a good statistic for the performance of the network.

When evaluating in general, images are fed to the network in batches of 8 images that are all classified simultaneously. Figure 7 shows the expected pipeline of data being fed to the network in batches.



```
(209_proj2_py37) bryanbed@bryanbed-VirtualBox:~/Documents/PyTorch-YOLOv3$ python
3 test.py --weights_path weights/yolov3.weights
Namespace(batch_size=8, class_path='data/coco.names', conf_thres=0.001, data_con
fig='config/coco.data', img_size=416, iou_thres=0.5, model_def='config/yolov3.cf
g', n_cpu=8, nms_thres=0.5, weights_path='weights/yolov3.weights')
Compute mAP...
Detecting objects: 100%|██████████████████████| 625/625 [48:31<00:00,  4.66s/it]
Computing AP: 100%|█████████████████████████████| 80/80 [00:02<00:00, 31.35it/s]
Average Precisions:
+ Class '0' (person) - AP: 0.690716111761301
+ Class '1' (bicycle) - AP: 0.4686961863448047
+ Class '2' (car) - AP: 0.5847853545047815
+ Class '3' (motorbike) - AP: 0.6173425471546101
+ Class '4' (aeroplane) - AP: 0.7368216071089109
+ Class '5' (bus) - AP: 0.7521598809811552
+ Class '78' (hair drier) - AP: 0.11363636363636365
+ Class '79' (toothbrush) - AP: 0.2643722437438991
mAP: 0.5145200189095042
```

Figure 7: YoloV3 object detection pipeline overview

3.2. Inference Accuracy

Once compiled, the Yolo-V3 repository has an evaluation test, which determines the mAP statistic across the entire test dataset. We ran three different iterations of this evaluation in order to compare the accuracy of our network, with and without different forms of quantization. The list below describes our test variations.

1. Non-quantized Yolo-V3 network - **As Published** [1]
2. Non quantized Yolo-V3 network - **Test Run, Recompiled** with TVM and course code base
3. Quantized Yolo-V3 network - **Built-in Tensorflow Function** for floating point quantization function (tf.quantization.fake_quant_with_min_max_args)
4. Quantized Yolo-V3 network - **Custom Quantization Function** for 8/16 bit floating point conversion

The table below summarizes our evaluation results results:

Quantization Status:	**Resultant mAP**
As Published	0.5550
Test Run, Recompiled :	0.5150
Built-In Tensorflow Function (TVM .pb load)	0.1105
Custom Quantization Function (TVM .pb load)	0.0882

From the results above, we can see that we take a significant hit to object detection accuracy when either of the quantization functions are run on the Yolo-V3's original weights, biases and feature map values. With an original mAP of 0.555 as published in the paper, and 0.515 after recompiling and running on our local systems, we see an over 80% hit to accuracy when quantizing using both our custom function and the built-in Tensorflow function. It is important to note that this accuracy hit comes without any additional training, and is being assessed using an evaluation criteria that has a severe cutoff point at 50% intersection over union. Therefore, we believe that what is happening here is that the quantization functions are cutting the accuracy of the bounding box stage of the Yolo-V3 network, to the point where IoU is less than 50% and the classification is deemed completely incorrect, even if the label is correct.

3.3. Quantization Accuracy

To analyze the difference in quantization accuracy between our custom quantization function, and the built-in Tensorflow function, we performed additional analysis on the accuracy of these algorithms in quantizing random values between the min and max for an 8-bit float with three exponential bits. We ran the quantization functions on identical 64x64x64 arrays containing values generated through an random distribution between -15.5 to +15.5, passed them through the 'difference' function. Results are shown below:

Quantization Method:	**Difference Across Sample 64x64x64 array**
No Quantization	0.00000
Built-In Tensorflow Function	0.00494
Custom Quantization Function	0.01191

The results above show that we can describe the difference in evaluation results between the built-in tensorflow function and our custom quantization function directly. We believe that this different is a result of some edge cases or some liberties that could have been taken in the Tensorflow implementation, which does another form of symbolically converting values.

3.4. Compression Analysis

The motivation for quantizing weights, biases and feature map values is primarily to reduce the size of the resultant network without demolishing the accuracy of the network. We believe that with ample retraining, we would be able to achieve network accuracy on scale with the original accuracy. Additionally, we would expect a quantization scheme that actually used 8 and 16 bit alternatives for the values, rather than symbolic representations as 32-bit floats, we would be able to vastly reduce the size of the network. The table below shows the results of xzip, gzip and bzip compression algorithms on .pb files generated from no quantization, built-in Tensorflow quantization function, and our custom quantization function.

Quantization Status:	**no compression**	**bzip**	**gzip**	**xzip**
Recompiled, Original:	247.8MB	237.7MB	230.9MB	229.0MB
Built-In Tensorflow Function:	247.8MB	237.7MB	230.9MB	229.1MB
Custom Quantization Function:	248.7MB	**37.4 MB**	**55.1 MB**	**36.5 MB**

From the results above, we can see that even though we received better evaluation results, initially without retraining, from the built-in Tensorflow functions, we ultimately get much better compression from our custom function. As advertised, the Tensorflow function uses a symbolic conversion as well, but from these results we see that common compression algorithms like bzip, gzip and xzip fail to compress the networks weights, biases and feature map values when quantized using the built-in API functions.

Instead, when using our custom quantization function, implemented in 'tf_symbolic_convert', we see clear success in compression, reducing the representative tensor graph by over 6.81x in the case of xzip, 6.6x for bzip and 4.51x in the case of gzip compression. These results clearly show that our quantization function is working, as the compression algorithms are easily able to store 32-bit float values that comprise much of the networks parameters, into smaller representations without loss.

4. Conclusion

We have determined from the exercises reported here that the Yolo-V3 network can be successfully quantized and compressed for embedded applications. We see significant accuracy loss when quantizing weights, biases and feature map values from 32-bit bit float to 8-bit float for weights and feature maps, and 16-bit float for bias values. However, this result does not include potential benefits from retraining. Typically, when altering the weights and biases of a network, the initial results in evaluation are poor, and tend to rise over time. The accompanying report to those one, titled: 'Project3 - Yolov3 Network Pruning' continues on the analysis in this report, by pruning finite weights and biases from the network and retraining to show how network evaluation initially drops but improves over time with subsequent training.

References

[1] R. G. A. F. Joseph Redmon, Santosh Divvala, You only look once, unified, real-time object detection, CVPR.
URL https://www.mdpi.com/2072-4292/11/11/1363

[2] Wikipedia, Single-precision floating-point format.
URL https://en.wikipedia.org/wiki/Single-precision$_{f}loating-point_{f}ormat$

[3] D. W. T. Verts, A 8-bit floating point representation.
URL http://www.cs.jhu.edu/ jorgev/cs333/readings/8-Bit$_{F}loating_{P}oint.pdf$

[4] A. Cherkaev, The secret life of nan.
URL https://anniecherkaev.com/the-secret-life-of-nan