Lighter, Faster Semantic Segmentation by Post-Quantization and Quantization-Aware Training

CS256 GroupE members: Anand Vishwakarma, Inhee Park, Sagar Shahi, Sherif Elsaid, Sriram Priyatham Siram

Abstract:

Deep Learning models require a lot of computational power and memory space to perform accurate inferences. Quantization of these models would reduce the model size and computation complexity. The main benefits of quantizing the models are reduced cloud computing costs, reduced latency, security, and privacy as the computations are all performed on the edge devices. In this project, we quantized DeepLab V3 plus using the quantization tools provided by tensorflow. We are able to use both quantization aware training and post training quantization to reduce our model-size and improve the inference time. The post training quantized model was however not able to perform any inference but the quantization aware training model was able to produce good results for data that resembled the training data. Although the model was able to produce the inferences, the prediction accuracy was greatly reduced. The purpose of our project is to use tools available in tensorflow to quantize DeepLab V3 plus.   Finally, we collected all the models (float32, quantized unit8) and obtained inference segmentation images, which is organized in the Jupyter notebook. (https://github.com/SherifSabri/ImageSegmentationWithDeeplab/blob/master/inference_deeplab.ipynb)

I. INTRODUCTION

Problem Statement:

Machine Learning Models are getting larger while the need to use them on edge devices like smartphones and IOT hardware is becoming more important. Extremely long time periods for model training, security concerns about sending the user data to a remote server for processing and inference, coupled with the large output model size that makes OTA updates difficult, are some of the reasons that are driving the need to find a more efficient way for training neural networks. And once a model has been created, the memory and compute constraints on those edge devices derive the need to find a more efficient way for running the inference/prediction on the trained model.

For those reasons and more, researchers and machine learning engineers have been developing different algorithms and methods for improving the efficiency of deep learning techniques.

Those approaches can be divided into two broad categories:

- Improving the efficiency of the hardware running the ML models (outside the scope of this paper).
- Improving the efficiency of the algorithms training the ML models and using Model Compression techniques for efficient inference and smaller model size.

Some of the approaches in the second category include:

- **Quantization** (the main focus of this paper): By using multiple techniques such as clustering the weights by reducing the size of bits of floating-point precision down to 8-bit integers. Or by using Huffman Encoding where the more frequently used weights would have larger bits to represent their values. Or by doing the whole training using floating points for the weights and activations and reverting to integers only during the inference/prediction stage.
- **Pruning**: Removing redundant weights from the network that does not have a direct impact on network performance.
- **Architecture Design Strategies**: by re-designing and using different encoder, and decoder networks (Resnet50, Resnet101, SqueezeNet, SqueezeNext, MobileNets, DeepLab, ...etc) and restructuring them in different ways to find the optimal configuration.
- **Knowledge Distillation**: By transferring the knowledge from the large cloud network to the smaller edge network.
- **And other methods** such as Low-Rank Approximation, Using Binary net, and Winograd transformation.

Objective:

Using various quantization techniques and by adjusting and experimenting with the different hyperparameters, both during and after the training, we aimed to produce a tensorflow quantized model with comparable accuracy, smaller size, and faster inference time, than the models provides as an example by the original deeplab team.

We choose to carry out our research using “MobileNet v2” as the backbone network due to its state of the art performance and excellent benchmarks on a variety of tasks and the small size and lower latency of its produced models, we did the majority of the training on the pascal visual object classes datasets, along with cityscapes, and the ade20K datasets.

DeepLab v3+ : Deep Labelling for Semantic Image Segmentation

For the Decoder network, we choose Deeplab v3+.

DeepLab is a state-of-art deep learning model for semantic image segmentation, where the goal is to assign semantic labels to every pixel in the input image. It’s being developed and maintained by google, it has a proven track record, and is the technology behind some of the advancement on google’s pixel smartphone photography in recent years.

Deeplab's architecture consists of an encoder (extracting essential information via a convolutional network such as DCNN, ResNet, and Xception, downsampling, pooling) and a decoder (reconstruction/refinement of extracted information from encoding phase with upsampling, post-processing with CRF).

One core feature among DeepLab, DeepLab V3 and DeepLabV3+ is an **atrous convolution**: y[i] = sum_k x[i + r.k] w[k], where y is output, x is input feature, r is an atrous rate and w is a filter. Convolving input x with unsampled filters by inserting r-1 zeroes between two consecutive filters (if r = 1 is a standard convolution) and using different rate extends a field-of-view for denser feature extraction.

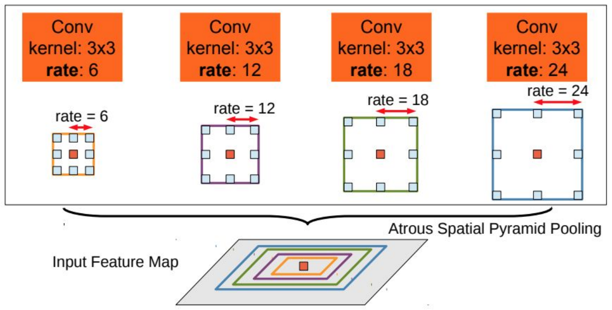


Fig. 4: Atrous Spatial Pyramid Pooling (ASPP). To classify the center pixel (orange), ASPP exploits multi-scale features by employing multiple parallel filters with different rates. The effective Field-Of-Views are shown in different colors.

II. Literature Survey

We reviewed many closely-related techniques developed for Quantization. There were two types of techniques used for Quantization: quantization aware training and post training quantization. Quantization aware training and post training quantization are similar in the sense that they both reduce the precision of the weights but different in the sense that one is used during the training process and another is used after training is completed.

Quantization Aware Training. There are many methods used for Quantization-aware training. Some of the most effective methods are Binary Connect (BC), BinaryNet, and Binary Weighted Network(BWN). These methods are based on the principles of Expectations BackPropagation and are closely related to each other. They only differ on how and where the quantization is applied to the weights.

Binary–Connect (BC). Binary-Connect performs quantization on the weights by converting all the real-valued weights into -1 or +1 based upon a function. It is performed during forward-propagation and backward-propagation but not for the parameter updates. Depending upon the types of function used BC are of two types. Deterministic Quantization is done when the Quantized Weights are calculated with the following formula.

$$w_b = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

Here, $_b$ is the quantized weight and is the actual weights.The quantized weights can also be calculated using the stochastic approach like below.

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w), \\ -1 & \text{with probability } 1 - p. \end{cases}$$

Here p is the probability assigned to the real weight and σ is the hard-sigmoid function given as:

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$$

Instead of x we use the value of weight for the neuron at the given level. BinaryNet is an extension on Binary-Connect. It uses all the same method of activation as that of Binary-Connect but it also performs binarization on the activations.

Binary–Weight–Network(BWN). BWN differs from BC on how the quantized weights are calculated. The main idea behind this method is the estimation of the real weights W based on the binary filter B and scaling factor α. B is a binary tensor that only contains -1 or positive 1 and α is a positive scalar. The weight is given by W = B * alpha. If I is the input and W is the weight, our regular convolution is represented by $I * W$ but our binarized convolution is $(I \oplus B)\,\alpha$. B is calculated in the same way as we calculate weights in the BC method but the main difference is how we calculate α. α is calculated by using the following formula on the weights of the neural network. $\alpha = \frac{1}{n}\|W\|$ at a given layer of the neural network.

BC and BWN significantly reduces the model size of the model but at the cost of accuracy. For AlexNet, the model size dropped from 475MB to 7.4MB. The accuracy dropped significantly for Top-1 and Top-5 classes for BC from 80.2 to 61.0 and 56.6 to 35.4 respectively. BWN performed better than BC as the accuracy was dropped from 80.2 to 77.0 and 56.6 to 53.8 for Top-5 classes and Top-1 class respectively with significant decrease in size of the model. Our approach is related to this method is a sense that we are also decreasing the range of the weights from float32 to UINT8 instead to binary values.

Post Training Quantization. The quantization of weights or/and activations on the trained model is called post training Quantization. The main idea behind this method is to convert higher precision numerical values to lower precision. For example, if our model weights and activation precision is float32, we can quantize it by reducing the precision from float32 to float16. The model sized will be reduced but at the cost of accuracy.

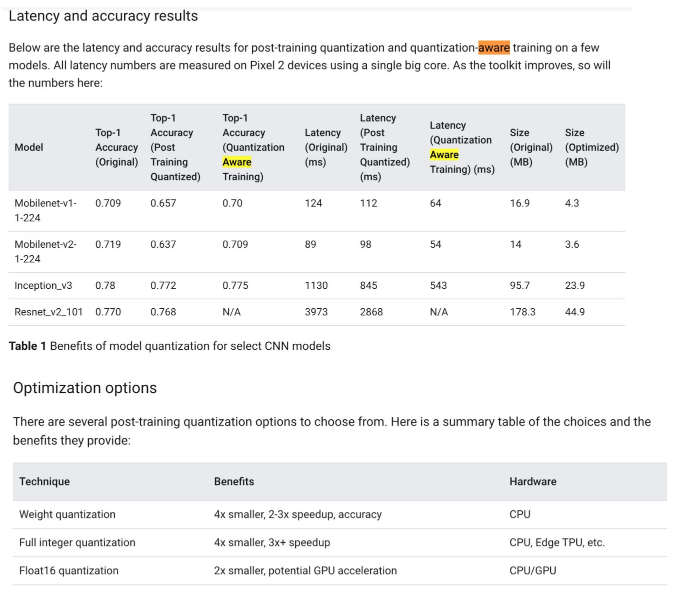
Our Approach. We used both quantization aware training and post training quantization techniques provided by tensorflow. In both techniques we reduced the precision from float32 to uint8 for the weights. As expected quantization aware training was able to produce better results than post quantization technique. Tensorflow provides useful apis for both quantization aware training and post training quantization of deep learning models with just a few extra lines of codes. The quantization aware training saves both quantized weights and real weights. The weight updates is done in the real weights based on the gradients of the quantized weights as in the formula below.

$$w_{float} = w_{float} - \eta \frac{\partial L}{\partial w_{out}} . I_{w_{out} \in (w_{min}, w_{max})}$$

$$w_{out} = SimQuant(w_{float})$$

Project Idea:

Using Deeplab V3 as the decoder network, and experimenting with different backbone networks, and by utilizing different parameters and techniques for a more efficient Quantization result, we plan to improve upon the currently used standards in model compression and achieve a result that proves to be faster, more efficient while sacrificing little in terms of accuracy and performance, we also hope to demonstrate our results in comparison with the default implementation provided by the deeplab original authors and show a noticeable improvement upon it.



Latency and accuracy results

Below are the latency and accuracy results for post-training quantization and quantization-aware training on a few models. All latency numbers are measured on Pixel 2 devices using a single big core. As the toolkit improves, so will the numbers here:

Model	Top-1 Accuracy (Original)	Top-1 Accuracy (Post Training Quantized)	Top-1 Accuracy (Quantization Aware Training)	Latency (Original) (ms)	Latency (Post Training Quantized) (ms)	Latency (Quantization Aware Training) (ms)	Size (Original) (MB)	Size (Optimized) (MB)
Mobilenet-v1-1-224	0.709	0.657	0.70	124	112	64	16.9	4.3
Mobilenet-v2-1-224	0.719	0.637	0.709	89	98	54	14	3.6
Inception_v3	0.78	0.772	0.775	1130	845	543	95.7	23.9
Resnet_v2_101	0.770	0.768	N/A	3973	2868	N/A	178.3	44.9

Table 1 Benefits of model quantization for select CNN models

Optimization options

There are several post-training quantization options to choose from. Here is a summary table of the choices and the benefits they provide:

Technique	Benefits	Hardware
Weight quantization	4x smaller, 2-3x speedup, accuracy	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, etc.
Float16 quantization	2x smaller, potential GPU acceleration	CPU/GPU

III. METHODOLOGY

To carry out the quantization experiments we have taken DeeplabV3+ on MobilenetV2 architecture as the base model and utilized TensorFlow's inbuilt support for quantization. We have formulated the experiments to cover both post training quantization and quantization aware training. We carried out the experiments on the pretrained model made available by Deeplab as well on our own models which we trained from scratch. All the experiments were carried on the same set of base parameters. Figure Q1 illustrates the flow of the experiments. We choose to perform these specific set of experiments, as they will allow us to compare and contrast the inference quality and accuracy of various quantization methods.

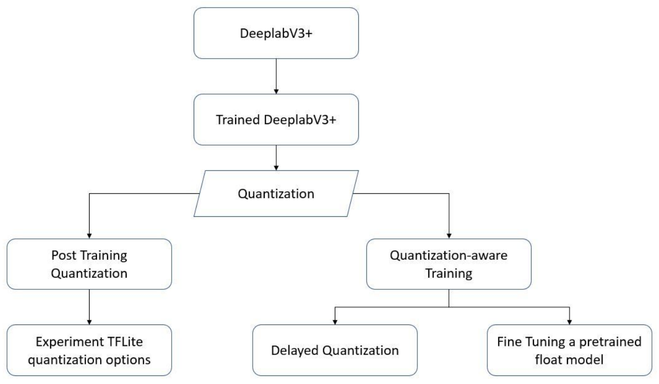

DeeplabV3+
Trained DeeplabV3+
Quantization
Post Training Quantization
Quantization-aware Training
Experiment TFLite quantization options
Delayed Quantization
Fine Tuning a pretrained float model

Figure. Q1

a. Quantization Aware Training:

1. **Fine Tuning a pretrained model**: We have taken a model trained on PASCAL VOC dataset. Using the training scripts provided by Deeplab, we have fine tuned the model by retraining it using a small learning rate, a small number of steps and by passing the parameter "quantize_delay_step”. This parameter accepts the number steps after which the quantization aware training must be initiated. In this case, 0 has been passed to this parameter as the model is pretrained and we are training only to quantize the weights. We then exported the resultant model and converted it into a TFLite model using the command “tflite_convert”. We then ran inference on the resultant TFLite model.

2. **Delayed Quantization**: Delayed quantization is usually performed when you want to complete training and quantization in the same run. In this approach we configure the training to run for a specific number of steps and also pass a parameter to specify after which step of the training the quantization should start. Based on previous experiments if it is known that a model converges after N steps, then we delay the quantization until then i.e we set the delay parameter to N. We trained the DeeplabV3+ model from scratch for delayed quantization using the using the training scripts provided by Deeplab. The training has been carried out for 36000 steps, where the quantization aware training was performed only for the last 6000 steps. Once the model is trained, we followed the procedure described in the above subsection to generate a TFLite model and performed inference on the resultant model.

A high level flow of a quantization aware training is illustrated in figure Q2

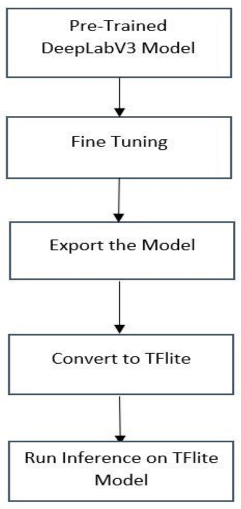
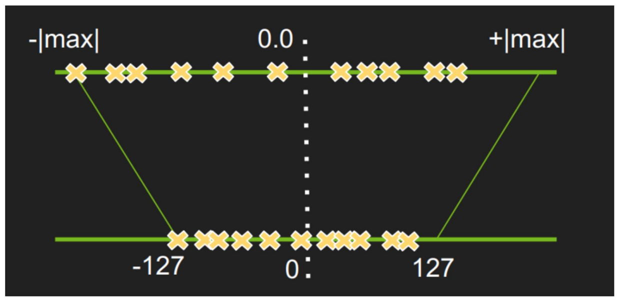


Figure. Q2

B. **Post Training Quantization:**

Post-quantization is a simple way of compressing the weights to lower precision, float32 to 8-bit, 16-bit, or 32-bit integers without extra training phase. No input data is required contrary to the quantization-aware-training. The quantization is done as a part of preparing a model for inference.

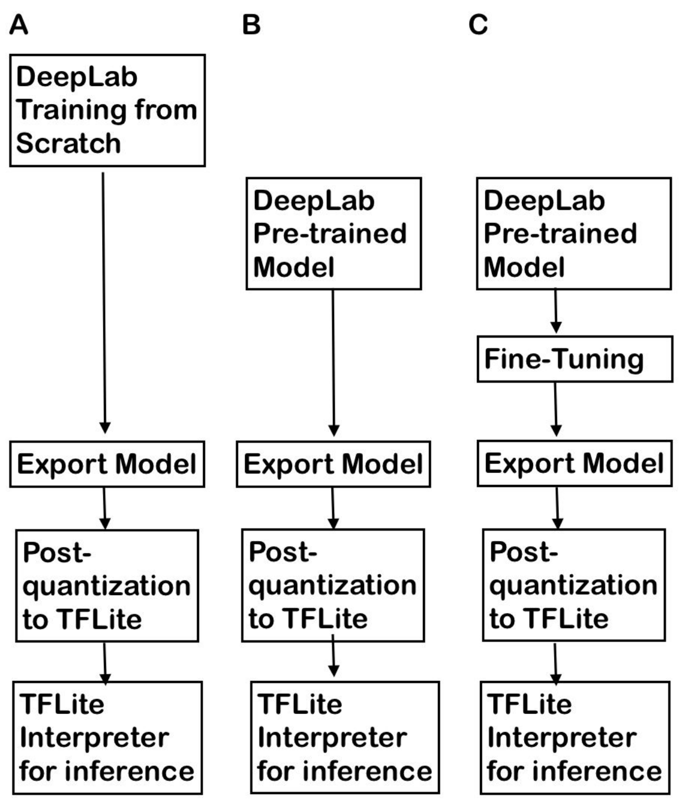


Tensor Values = FP32 scale factor * int8 array.

We tested the post-quantizations on the following three models:

A. Post-quantization based on our training from scratch
B. Post-quantization based on the pre-trained model without any fine-tuning
C. Post-quantization based on the pre-trained model with fine-tuning

High-level workflow is following:



A
B
C
DeepLab Training from Scratch
DeepLab Pre-trained Model
DeepLab Pre-trained Model
Fine-Tuning
Export Model
Export Model
Export Model
Post-quantization to TFLite
Post-quantization to TFLite
Post-quantization to TFLite
TFLite Interpreter for inference
TFLite Interpreter for inference
TFLite Interpreter for inference

Figure. Post-quantizations based on the three difference models

First step is to obtain a pb file for tensorflow format by specifying the check-point (`model.ckpt-###`).

```
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
python deeplab/export_model.py \
  --logtostderr \
  --checkpoint_path="/init_models/model.ckpt-30000" \
  --export_path="noFineTune_frozen_inference_graph.pb" \
  --model_variant="mobilenet_v2" \
  --num_classes=21 \
  --crop_size=513 \
  --crop_size=513 \
  --inference_scales=1.0
```

Next step is to convert the full float32 model (the weight frozen pb file) to quantized uint8 model using `tflite_convert`. We found that in order to carry out post-quantization, TensorFlow version must be >=15.0.0 (which has upgraded version of TensorFor Lite, thus , has correct version of `tflite_convert).` Otherwise, one of operations in the MobileNetV2 network, `conv_2d` is not supportive in older version of TensorFlow.

The `tflite_convert` command line has many quantization pertinent options. For the purpose of post-quantization, "`--post_training_quantize`" must be used, thus weights are quantized to 8-bit integer. For the post-quantization, the "--inference_type" must be FLOAT because the base model is float32 bit (i.e. not working for QUANTIZED_8BIT in this case).

```
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
tflite_convert \
--graph_def_file=noFineTune_frozen_inference_graph.pb \
--output_file=\
noFineTuye_frozen_inference_graph_postquant.tflite \
  --inference_type=FLOAT \
  --post_training_quantize \
  --output_format=TFLITE \
  --input_shape=1,513,513,3 \
  --input_arrays="MobilenetV2/MobilenetV2/input" \
  --std_dev_values=128 \
  --mean_values=128 \
  --change_concat_input_ranges=true \
  --output_arrays="ArgMax"
```

Once we obtained the models based on the above experiments, they were evaluated and the mean IOU scores have been recorded. The results are discussed in the Results section.

IV. IMPLEMENTATION DETAILS

For all the experiments, the DeeplabV3+ with MobileNet v2 backbone was used as the base model.

A. Tensorflow Script for Training From Scratch (Full float32 model)

The base deeplab model has been trained from scratch with the following parameters using the training scripts provided by Deeplab but removed the initial check point, thus enabled a training from scratch.

- training_number_of_steps = 100000
- Learning rate = 0.0001
- Dataset = PASCAL VOC
- Train Batch Size = 16
- Train Crop Size = 513, 513

B. Tensorflow Script for Quantization-Aware Training from Scratch (with Delayed Quantization > 0; UINT8)

The base deeplab model has been trained from scratch with the following parameters using the training scripts provided by Deeplab.

- training_number_of_steps = 36000
- quantize_delay_step = 30000
- base_learning_rate = 0.0001
- Dataset = PASCAL VOC

The training scripts provided by Deeplab internally utilize Tensorflow's contrib.quantize module which is used to create graphs for quantized training.
The generated model was exported into a frozen graph and converted into TFLite model using the *tflite_convert* command. Then inference was performed using TFlite Interpreter.

C. Tensorflow Script for Post-Quantization

Detailed scripts are shown in section III-B.

D. Tensorflow Script for Quantization-Aware Training from Pre-trained Model

A Deeplab V3+ model trained on PASCAL VOC dataset was fine tuned with the following parameters

- training_number_of_steps = 3000
- base_learning_rate = 3e-5
- quantize_delay_step = 0

As explained in section III, since the quantize_delay_step was set to 0, the quantization aware training will start from the first step itself. This is preferred in this case as we already have a converged pretrained model and we are just training for quantization. The resultant model is frozen and then converted into a TFlite model. The inference results are shown in the Results section.

E. All Models and All Inferences in Jupyter Notebook

The experiments which involved training were performed on an Amazon EC2 instance with the following configuration.

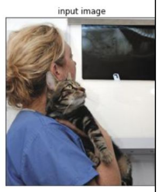
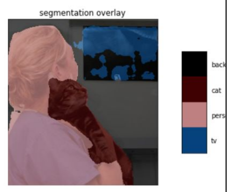
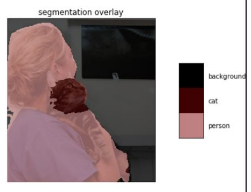
- Instance type: p2.xlarge
- AMI: Deep Learning AMI (Ubuntu) Version 24.2
- Virtual CPU count : 4 (2 Core * 2 Threads per core)

- Memory: 61 GB
- GPU count : 1

V. SEGMENTATION INFERENCE USING PRE-TRAINED MODELS

The DeepLabV3+ is a framework with a signature feature of Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation [DLv3+]. Also, the DeepLabV3+ is built on the various CNN architectures (MobileNet-v2 for faster inference, Xception_65 for better performance, Xception_71) as the backbone for extracting features. As a preliminary step to understand better the DeepLabV3+, we utilized the TensorFlow DeepLab Model Zoo [DLZoo]. There are three classes of pre-trained DeepLab models with several datasets, including (1) PASCAL VOC 2012, (2) Cityscapes, (3) MS-COCO and (4) ADE20K. Those models are prepared in the TensorFlow Deep Learning framework [TFModels]; 4 typical TensorFlow model files 1) `model-ckpt.meta`: contains graphDef that describes the data-flow, annotations for variables, input pipelines and other relevant information; 2) `model-ckpt.data-0000-of-00001`: contains all the values of variables including weights, biases, placeholders, gradients, hyper-parameters, etc.; 3) `model-ckpt.index`: describes the metadata of a Tensor; 4) `checkpoint`: records checkpoint information; and `frozen_inference_graph.pb`: restores weights inside a session.

Pretrained Models: Our aim of inference on the pre-trained models is tested with one image (not from train/val/test dataset) is to observe the difference in the resulting segmentation prediction over various pre-trained models. All the inference computation is carried out on the AWS, on the preconfigure instance of Deep Learning AMI (Ubuntu 16.04) by choosing p2.xlarge GPU. We can use already installed TensorFlow 14.0.0 (`source activate tensorflow_p36`), so that we can import needed modules. Implementation of the inference script is based on the Jupyter iPython script available in the GitHub, `deeplab_demo.ipynb` [DLgithub]. 1) We downloaded all the available DeepLab pre-trained models from model zoo [DVZoo]; 2) convert an input image to numpy array format by using PIL python module (Python Image Library); 3) inference core step by creating placeholders in TensorFlow as future variables to be determined at runtime by feed_dict argument during operations by `run(self, image)`; 4) resulting segmentation map, i.e. pixel-wise classified labels will be mapped onto the original pixel/image for visualization by `vis_segmentation(image, seg_map)`.
By simple visual inspection on the inference on single image, the most accurate prediction is from the pre-trained model Xception65 as a backbone architecture trained with MS-COCO and VOC 2012 dataset. Perhaps because the Xception65 based model is 439MB with mIOU of 87.8%, whereas MobileNetV2 is more than x20 smaller (23MB) with mIOU of 80.3%, thus Xception model may have more detailed information, resulting in higher accuracy segmentation prediction. Appendix E shows the inference result using the cityscapes dataset based pre-trained model.

Input Image	Xception	MobileNetV2
		

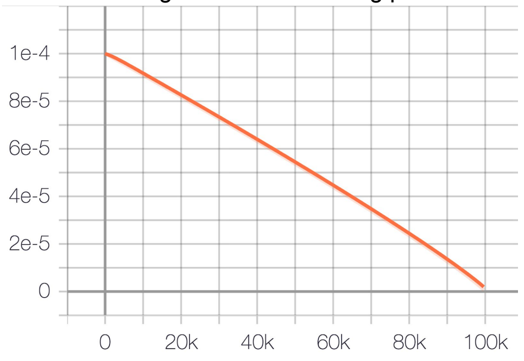
VI. EVALUATIONS

Prior to quantization, we evaluated full 32-bit float models in terms of mean IOU of 21 classes of the validation dataset as well as the overall mIOU for the following three cases:

A. Train-from-scratch (100,000 iterations; took 18 hours; overall miou 3.5%)

Owing to the relatively small dataset of PASCAL VOC 2012 as well as light model of MobileNetV2, we could train the model with the dataset from scratch on the AWS using 60GB RAM on a GPU. The following is a script for training. Overall 100,000 iterations took 18 hours.

And the learning rate overall training phase is following:



We then evaluated the model that we trained.
However, after evaluation, we found that the overall mIOU was too low about 3.5%. We found the github DeepLab Q&A that training the DeepLab from scratch using PASCAL VOC 2012 dataset is difficult, may expect to have better mIOU training with augmented dataset with COCO. While VOC 2012 dataset has 21 classes, whereas COCO dataset has 330K images with >200 labels. Perhaps training may require more training dataset. The following is the evaluation results on the 21 classes.

Class	MIOU
0	0.73
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

10	0
11	0
12	0
13	0
14	0
15	5.39e-07
16	0
17	0
18	0
19	0
20	0

Overall MIOU = 0.035

B. Fine-tuning (1000 iterations; overall mIOU=74%; if we used 10 iterations only, its overall mIOU=76%)

Class	MIOU
0	0.94
1	0.84
2	0.34
3	0.84
4	0.64
5	0.74
6	0.93
7	0.87
8	0.90
9	0.38
11	0.60
12	0.84
13	0.87
14	0.82
15	0.0.78
16	0.53
17	0.83
18	0.55
19	0.88
20	0.50

Overall MIOU:-0.74

C. Quantization-Aware Training (2491 iterations; overall mIOU=66%)

Class	MIOU
0	0.91
1	0.78
2	0.35
3	0.73
4	0.60
5	0.64
6	0.88
7	0.80
8	0.80
9	0.30
11	0.44
12	0.76
13	0.80
14	0.68
15	0.78
16	0.48
17	0.70
18	0.45
19	0.73
20	0.51

Overall MIOU = 0.66

VII. INFERENCE VISUAL INSPECTION

Post-quantization model inference:
Once the quantized model is generated (i.e. `*.tflite`), we can carry out inference on the quantized model using the TensorFlow Lite's Interpreter module to read information from the model such as input_details.

```
import tensorflow as tf
# quantized model without fine-tuning from pre-trained model
tflite_file="noFineTuyne_frozen_inference_graph_postquant.tf
lite"
interpreter=tf.lite.Interpreter(model_path=tflite_file)
interpreter.allocate_tensors()
input_details=interpreter.get_input_details()
output_details=interpreter.get_output_details()
print(input_details)
```

However, when we looked into input_details from the quantized model, we found that the quantization field is 0, which is not 0 for the quantization-ware-training.

```
[{'name':   'MobilenetV2/MobilenetV2/input',   'index':   6,
'shape': array([  1, 513, 513,   3], dtype=int32), 'dtype':
<class 'numpy.float32'>, 'quantization': (0.0, 0)}]
```

We then feed image pixel information (resize to 513x513, convert it to 1-column vector) to the quantized model based Interpreter as following:

```
import numpy as np
from PIL import Image
h=input_details[0]['shape'][1]
w=input_details[0]['shape'][2]
Image.open("/content/ADE_train_00019515.jpg").resize((h,w))
img=np.array(image)
input_data=np.expand_dims(img.astype(np.float32), axis=0)
```

Next, we set tensor type of index 6 (shown above from the input_details) to the input image data (input_data).

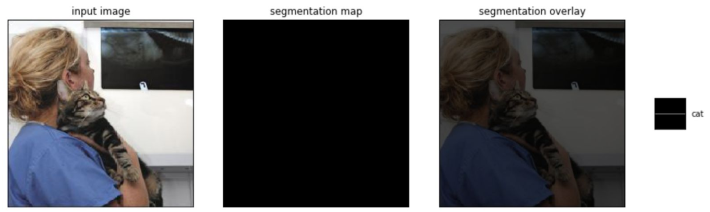
```
interpreter.set_tensor(input_details[0]['index'],
input_data)
interpreter.invoke()
output_data=interpreter.get_tensor\
(output_details[0]['index'])
results=np.squeeze(output_data)
```

However, resulting output data after assigning tensor to the image turned out to be all zeros by printing out non-zero elements from the result

```
print(np.nonzero(results))
```

```
=>  (array([],  dtype=int64),  array([],  dtype=int64),
array([], dtype=int64))
```

Thus the inference segmentation is not properly segmented.

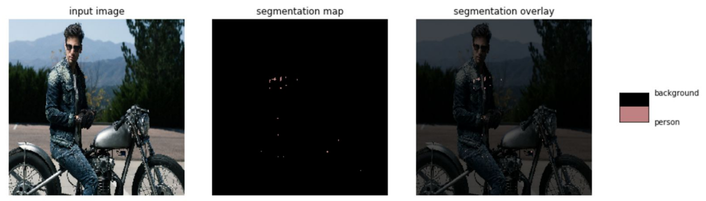


In order to find a potential bug, we tested the same workflow (exporting pb model, converting to tflite quantized model, all the way to inference using Interpreter) on the three different initial models (no fine tuning, fine-tuning, and training from scratch). But we couldn't obtain proper inference results on the quantized models.

Quantization Aware Trained Model with Delay:
This model trained on the pascal dataset with 36000 iterations and a delay of 30000 steps, produced a quantized model with the correct quantization field (STD, Mean)

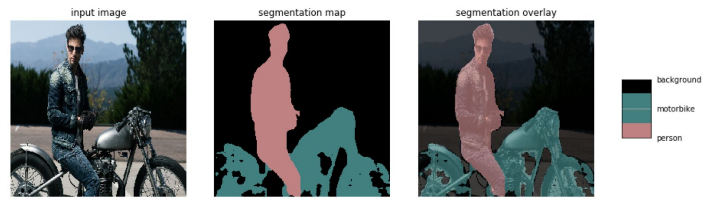
```
[{'name':   'MobilenetV2/MobilenetV2/input',   'index':   6,
'shape': array([  1, 513, 513,   3], dtype=int32), 'dtype':
<class 'numpy.uint8'>, 'quantization': (0.0078125, 128)}]
```

However the resulted inference was not better than the earlier attempt, with the model failing to adequately produce a successful segmentation mask.



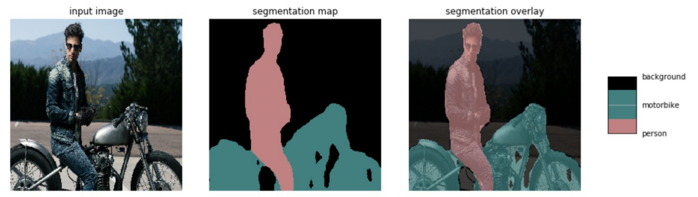
```
CPU times: user 681 ms, sys: 31.7 ms, total: 713 ms
Wall time: 785 ms
```

Post Quantization from Pre-trained Model:
We produced a quantized model by training the model using 3000 steps and no delay on a learning rate of 0.00003 and an output stride of 16, and using the same tf.lite converter with an std value and mean value of 128 each, we produced a working quantized model



```
CPU times: user 575 ms, sys: 24.7 ms, total: 599 ms
Wall time: 646 ms
```

In Comparison with the provided quantized model (.tflite) from the official quantized model zoo by the deeplab team, which had an mIOU of 74.26% and size of 2.2 MB



```
CPU times: user 586 ms, sys: 24.9 ms, total: 611 ms
Wall time: 637 ms
```

We can see that both the resulting segmentation mask, and inference time (both running on the same machine in the same notebook environment) are very comparable in a favorable way. Although we could not yet produce results that proved to be noticeably faster or smaller in size.

VIII. RESULTS AND DISCUSSION

By using two quantization techniques based on quantization aware training and post training quantization we are able to reduce the model by ¼ of the original model. The inference time was also much faster than the original model. The model accuracy however was reduced for the post training quantization model as it was not able to produce any inference at all. For our quantization aware trained model, we were able to produce better inferences for data that was closely similar to the data our training data but it didn't do well for the data that was not similar. In this project we only focused on the weight quantization and not activation quantization. In our future work we will explore if quantizing both weights and activations would yield better results. Also, for specific purpose we might need to train the model again using the training data that is closely related to our specific data. The evaluation results for PASCAL dataset produced by our quantized model are given below.

Through this project we also explored the possibility of improving the accuracy of a quantized model by retraining it. However, since a TFLite model cannot be retrained we could not make progress in that area. We will focus our future work in this direction.

REFERENCES

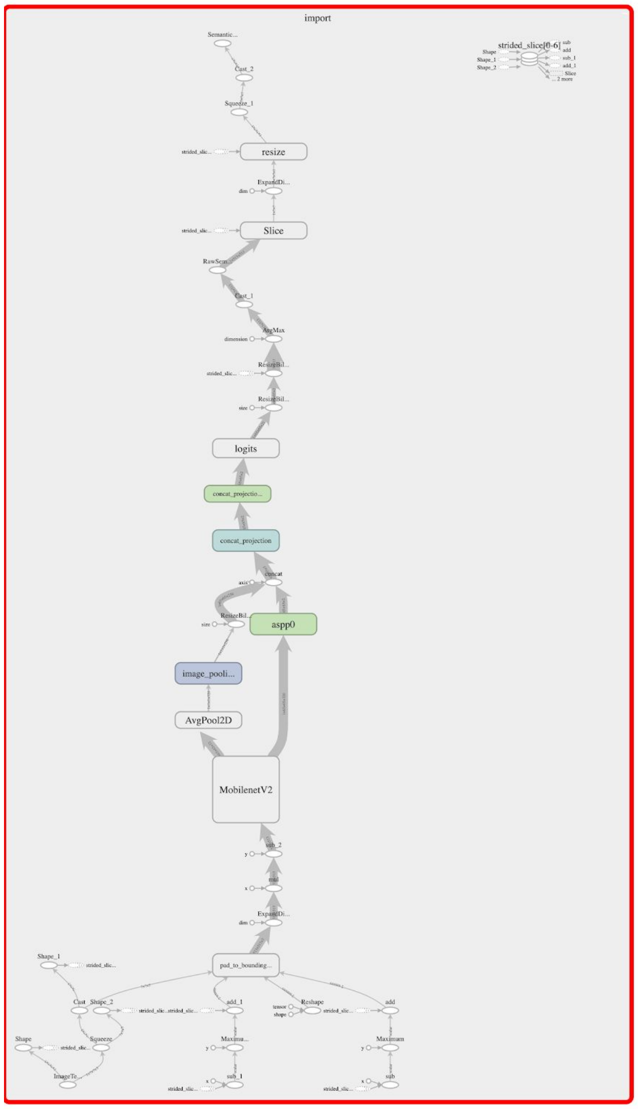
[1] Chen, Liang-Chieh, et al. "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation." *ArXiv.org*, 22 Aug. 2018, https://arxiv.org/abs/1802.02611.
[2] Chen, Liang-Chieh, et al. "Tensorflow/Models." *GitHub*, https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/quantize.md.
[3] Courbariaux, Matthieu, et al. "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations." *ArXiv.org*, 18 Apr. 2016, https://arxiv.org/abs/1511.00363.
[4] Courbariaux, Matthieu. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1" 2016,

https://pdfs.semanticscholar.org/123a/e35aa7d6838c817072032ce5615bb891652d.pdf
[5] Ouaknine, Arthur. “Deep Learning Model Compression for Image Analysis: Methods and Architectures.” *Medium*, Zyl Story, 6 Mar. 2018, https://medium.com/zylapp/deep-learning-model-compression-for-image-analysis-methods-and-architectures-398f82b0c06f.
[6] Rastegari, Mohammad, et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks.” *ArXiv.org*, 2 Aug. 2016, https://arxiv.org/abs/1603.05279.
[7] Krishnamoorthi, Raghuraman. “Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper.” *https://arxiv.org/pdf/1806.08342*, Google.com, 21 June 2018.

[Compression]

[DVZoo]
https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md
[TFModels] https://github.com/tensorflow/models
[VOC2012] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/
[Cityscapes] https://www.cityscapes-dataset.com/
[COCO] http://cocodataset.org/#home
[ADE20k] https://groups.csail.mit.edu/vision/datasets/ADE20K/
[DLgithub]
https://github.com/tensorflow/models/blob/master/research/deeplab/

Appendix A:


import
Semantic...
Cast_2
Squeeze_1
resize
ExpandDi...
Slice
RawSem...
Cast_1
ArgMax
ResizeBil...
logits
concat_projectio...
concat_projection
concat
aspp0
image_pooli...
AvgPool2D
MobilenetV2
sub_2
add
pad_to_bounding...
Shape_1
Cast
Shape_2
add_1
Reshape
Shape
Squeeze
Maximu...
Maximum
sub_1
sub
strided_slice[0-6]