

Lighter & Faster Semantic Segmentation by Post-Training Quantization & Quantization-Aware Training



SJSU CS 256 GroupE

Anand Vishwakarma, Inhee Park, Sagar Shahi, Sherif Elsaid, Sriram Priyatham

overview

1. Computer Vision Topic -- Semantic Segmentation

=> Hard problem: object detection + multi-classification at pixel level

2. Segmentation Framework -- DeepLabV3+

=> Atrous convolution (faster), Encoder-Decoder (shaper boundary)

3. Backbone Architecture -- MobileNetV2

=> Depthwise-separable-convolution + Inverted-Residual-Layer; TFLite for Mobile & IoT

4. Project Task -- Quantization

=> Quantization makes DeepLabV3+ on MobileNetV2 more lighter and faster

5. Quantization Approach 1 -- Post-Training-Quantization => F32 model prep -> Quantize

6. Quantization Approach 2 -- Quantization-Aware-Training => F32 model prep -> Quantize

7. Inference and Evaluation with Demo (Quantization & Inference Script on Colab)

=> Comparison between F32 vs. 8-bit integer inferences

1. Computer Vision Topic -- Semantic Segmentation

=> Hard problem: object detection + multi-classification at pixel level

Semantic Segmentation Requires

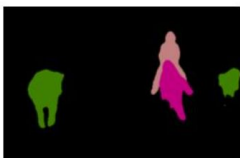
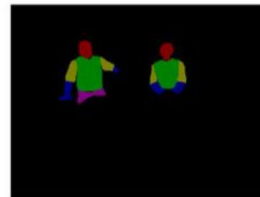
- 1) pinpointing the outline of objects;
- 2) imposing much stricter localization accuracy than image-level classification

PASCAL VOC 2012 Dataset for Semantic Segmentation

Pixel-by-pixel outline & classification for the given concepts/semantic labels



background
airplane
bicycle
bird
boat
bottle
bus
car
cat
chair
cow
dining table
dog
horse
motorbike
person



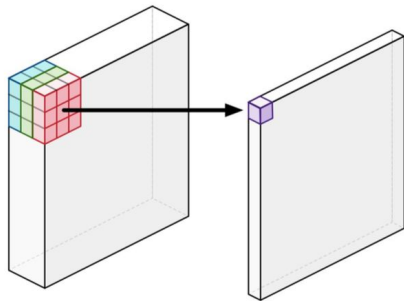
Head
Torso
Right foot
Right hand
Right lower arm
Right lower leg
Right upper arm
Right upper leg
Mouth
Neck
Nose
Right ear
Right eye
...

2. Segmentation Framework -- DeepLabV3+

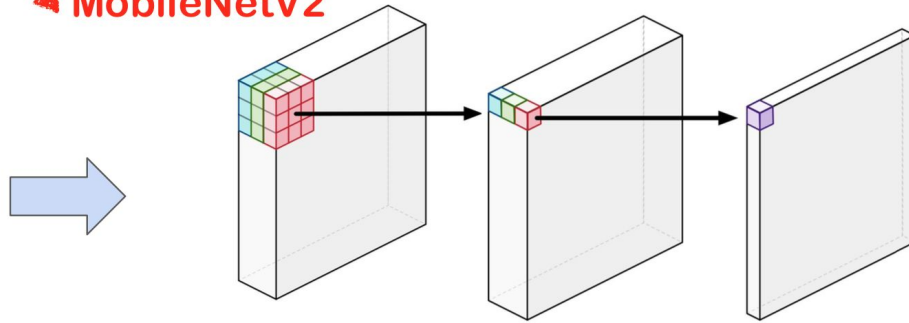
=> Atrous convolution (faster), Encoder-Decoder (shaper feature extraction)

DeepLabV3+: *Faster* “Atrous Depthwise Separable” Conv. on MobileNetV2

Regular Convolution

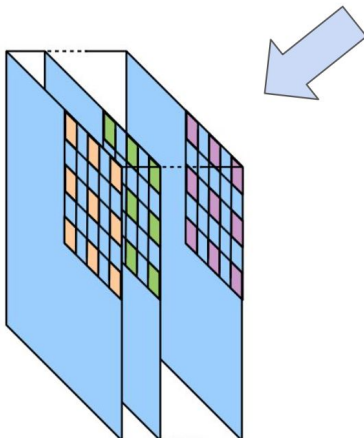


Xception: Depthwise Separable Conv.
MobileNetV2



**Fewer parameters,
thus less overfitting
and faster computation**

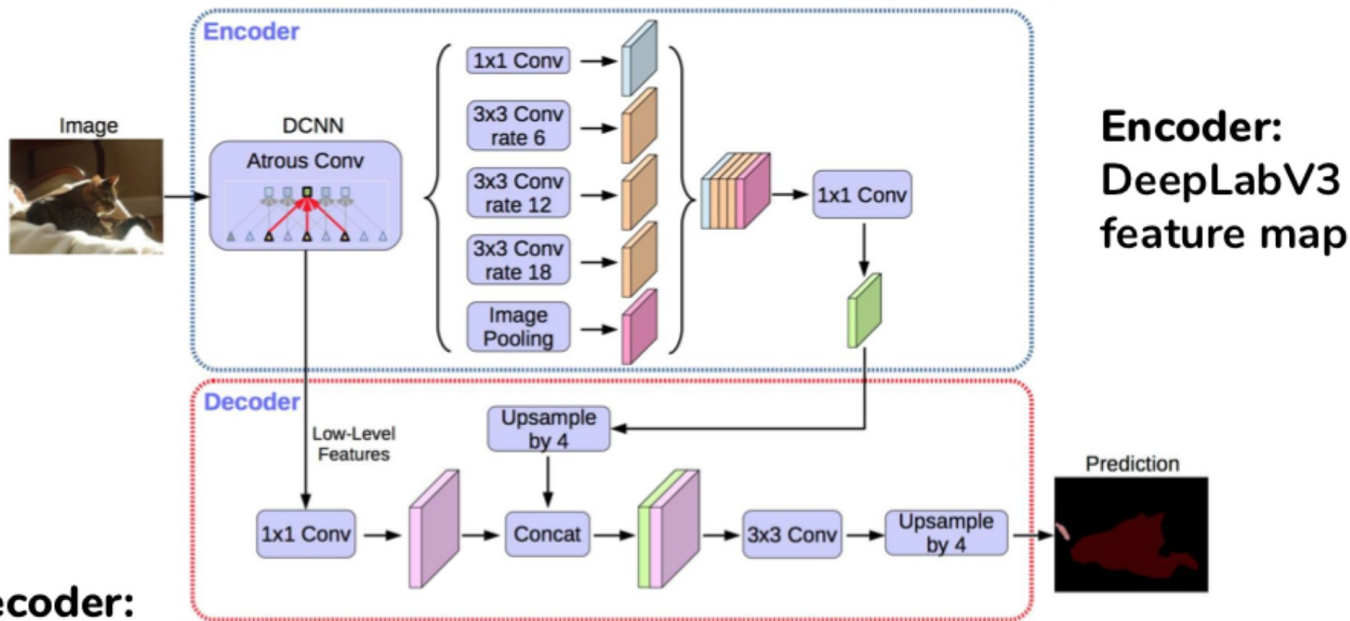
**“Atrous” Depthwise
Separable Conv.**



DeepLabV3+

DeepLabV3+: *Sharper* Boundary Segmentation by Encoder-Decoder

DeepLabV3+: Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation



Decoder:

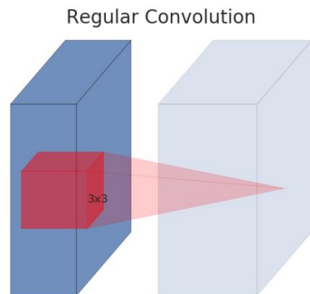
- (1) 1×1 conv for reducing channels of low-level feature map from encoder
- (2) 3×3 conv for sharper segmentation

3. Backbone Architecture -- MobileNetV2

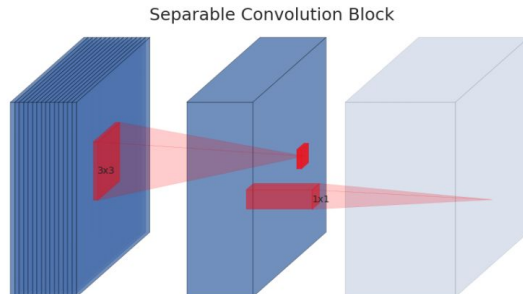
=> Depthwise-separable-convolution + Inverted-Residual-Layer; TFLite for Mobile & IoT

MobileNetV2: Depthwise-Separable-Conv + Inverted-Residual Layer

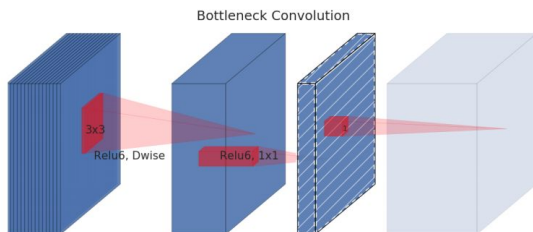
(a) Regular



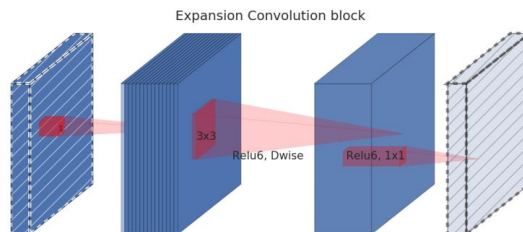
(b) Separable



(c) Separable with linear bottleneck

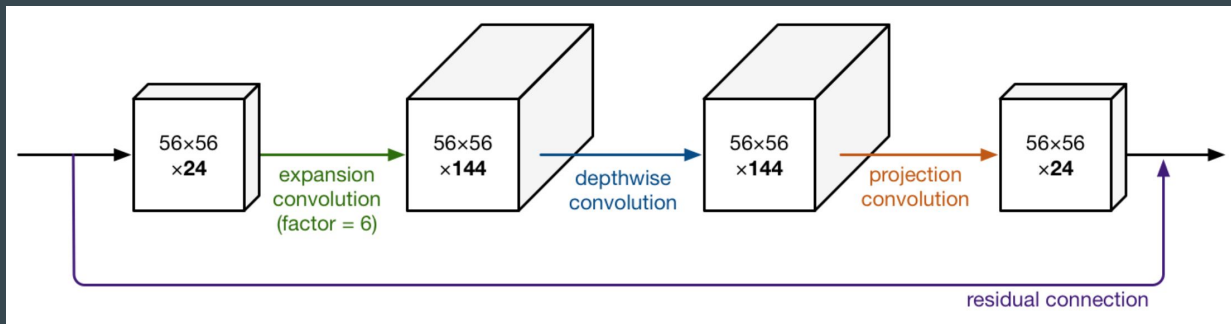


(d) Bottleneck with expansion layer

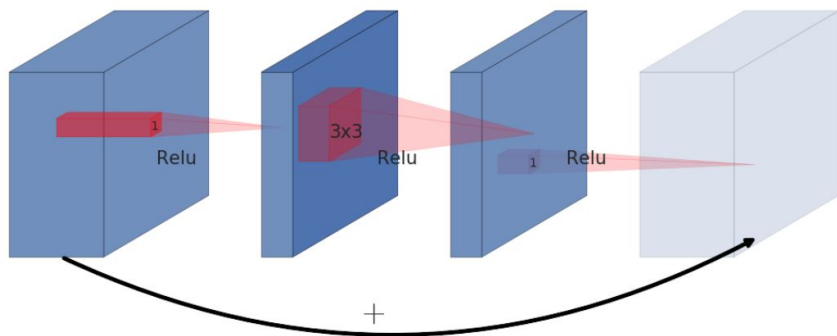


MobileNetV2:

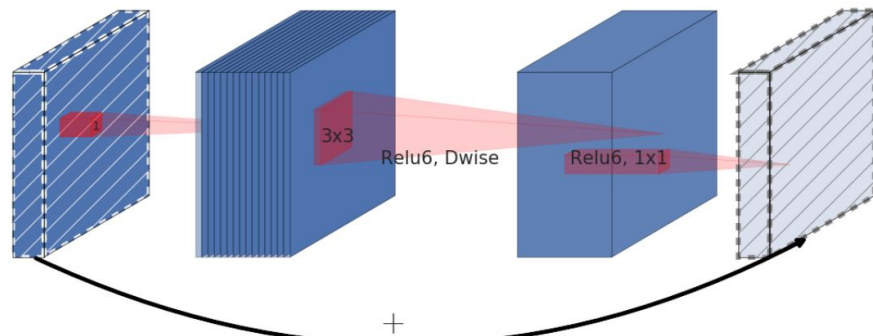
- 1) Depthwise-Separable-Conv
Like DeepLabV3+
- 2) Inverted-Residual Layer
Like ResNet but *Expansion*



(a) Residual block



(b) Inverted residual block



MobileNetV2 using TensorFlow Lite for Mobile & IoT

Build TensorFlow Lite

Build for iOS

Build for ARM64

Build for Raspberry Pi



TensorFlow

Install

For Mobile & IoT

Overview

Guide

Examples

Performance

Best practices

Benchmarks

Model optimization

Post-training quantization

Post-training weight quantization

Post-training integer quantization

Post-training float16 quantization

4. Project Task -- Quantization

=> Quantization makes DeepLabV3+ on MobileNetV2 more lighter and faster

Fixed point Fake quantization implemented in TensorFlow

1. First find the range of values in our tensor. The range we use is always centered on 0, so we find m such that

```
m = max(abs(input_min), abs(input_max))
```

2. Our input tensor range is then $[-m, m]$

3. Next, we choose our fixed-point quantization buckets, $[\text{min_fixed}, \text{max_fixed}]$.

4. With input tensor T , $\text{num_bits} = \text{sizeof}(T) * 8$

5. If T is unsigned, the fixed-point range is

```
[min_fixed, max_fixed] = [0, 2^(num_bits) - 1]
```

6. From this we compute our scaling factor,

```
s = (max_fixed - min_fixed) / (2 * m)
```

7. Now we can quantize the elements of our tensor

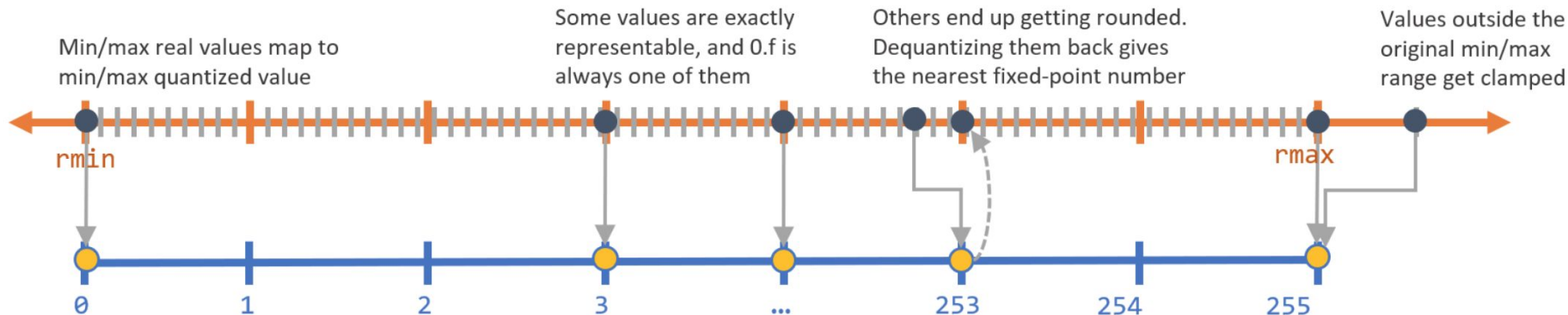
```
result = round(input * s)
```

Fixed point Fake quantization implemented in TensorFlow

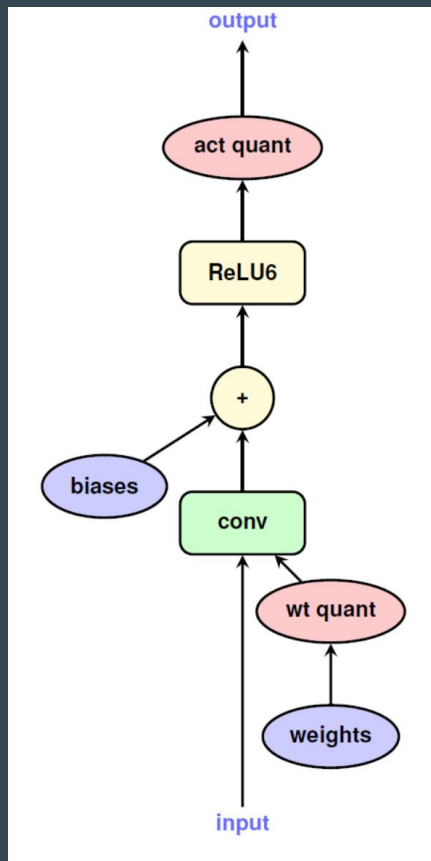
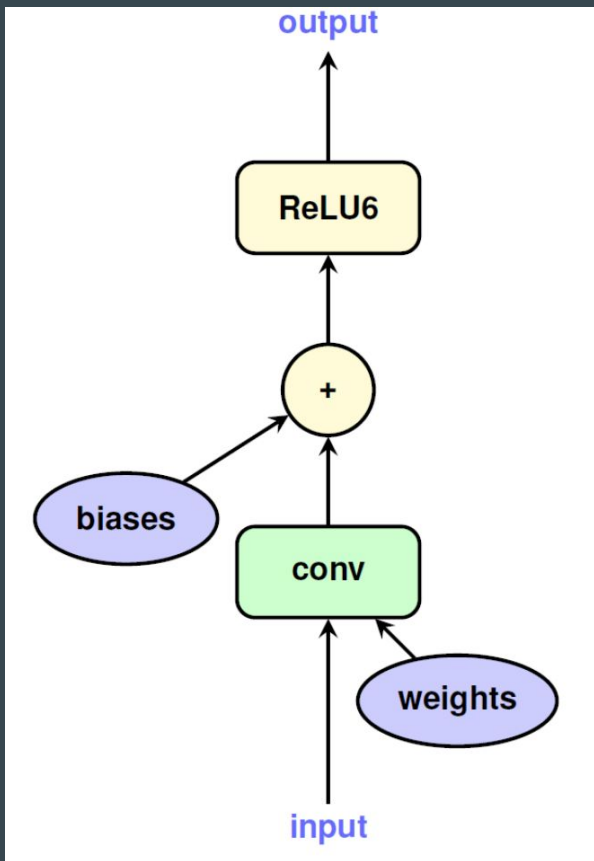
$$\begin{aligned} r &= \frac{r_{max} - r_{min}}{(2^B - 1) - 0} \times (q - z) \\ &= S \times (q - z) \end{aligned}$$

Here,

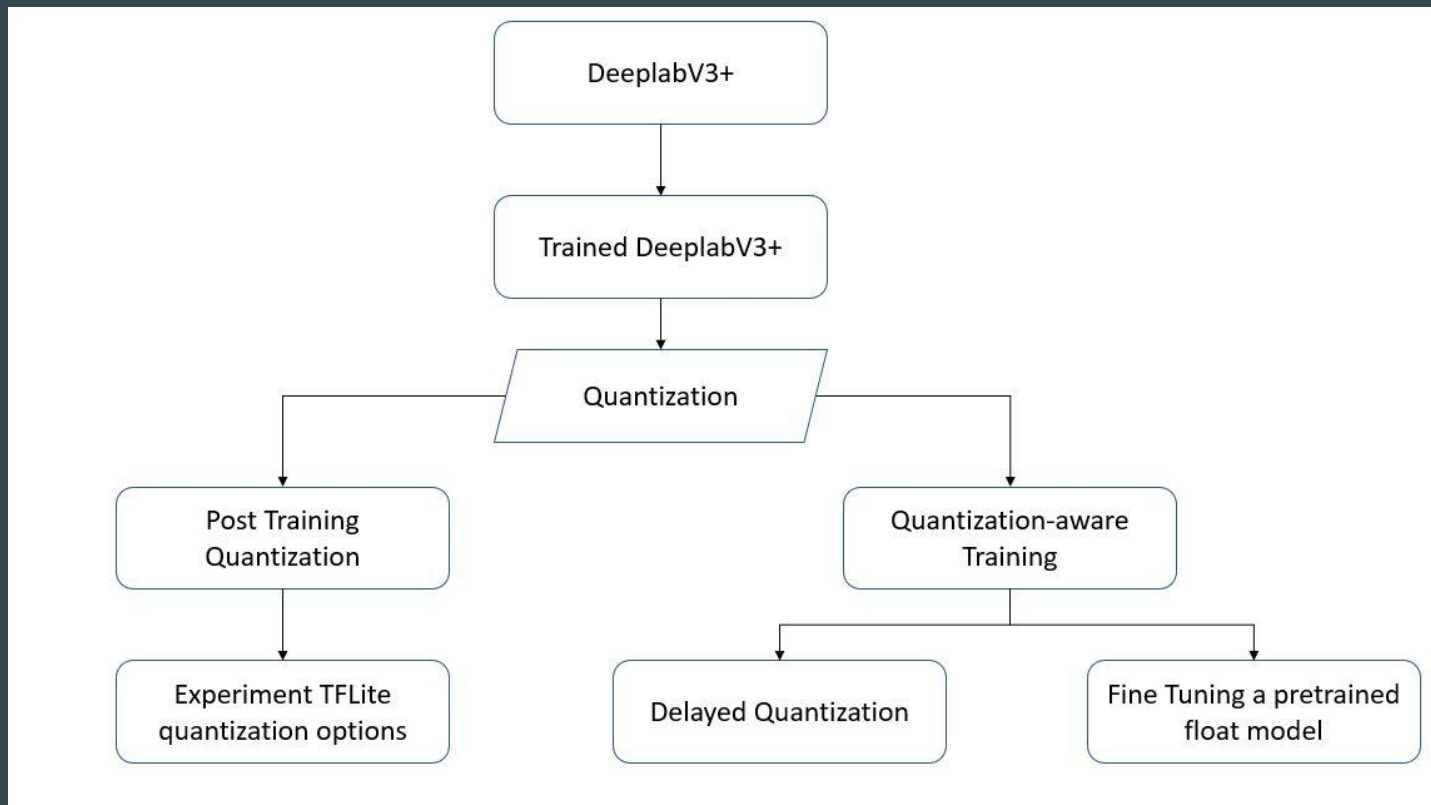
- r is the real value (usually `float32`)
- q is its quantized representation as a B -bit integer (`uint8`, `uint32`, etc.)
- S (`float32`) and z (`uint`) are the factors by which we scale and shift the number line. z will always map back exactly to `0.f`.



Fixed point Fake quantization implemented in TensorFlow



Quantization Experiments



5. Quantization Approach 1 -- Post-Training-Quantization

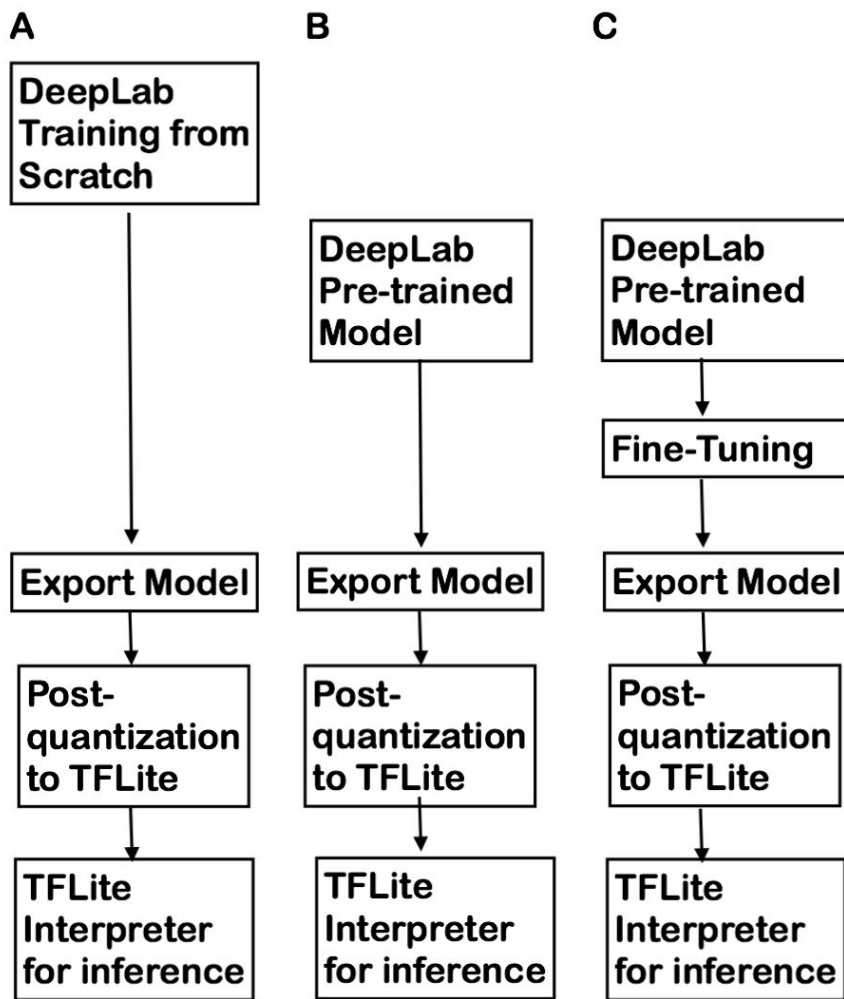
=> F32 model prep -> Quantize

Post-Training-Quantization Workflow

A. Train from scratch

B. No Fine-tuning

C. Fine-tuning
w/ 1, 10, 100, 1K, 10K iterations



Prep for Full 32-bit Float Models for Post-Quantization

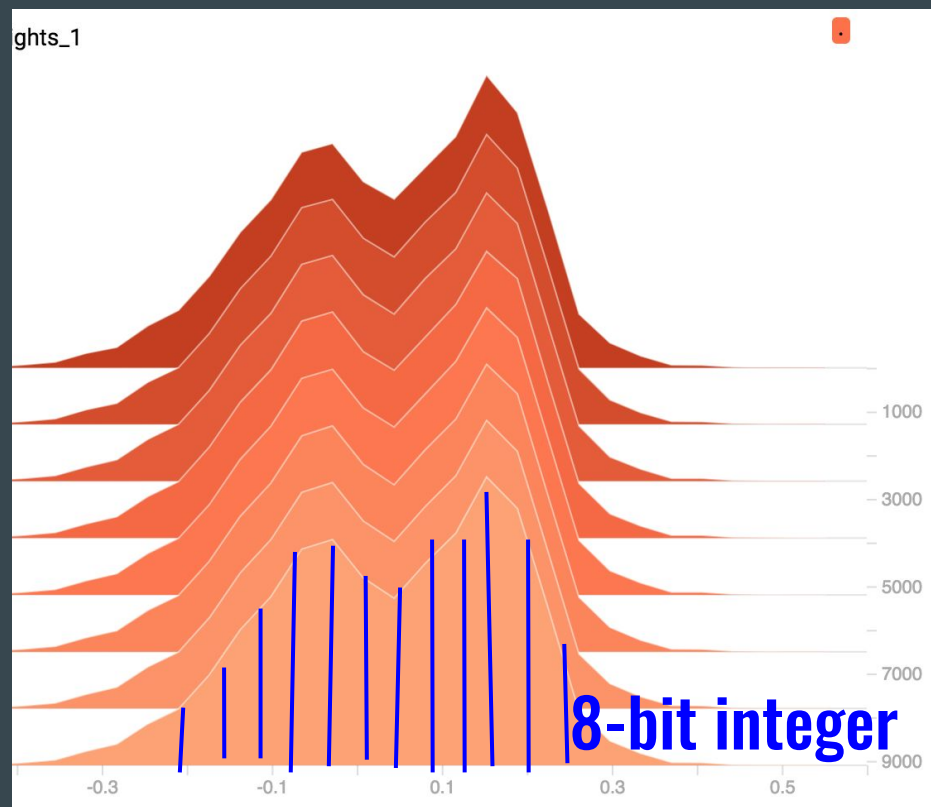
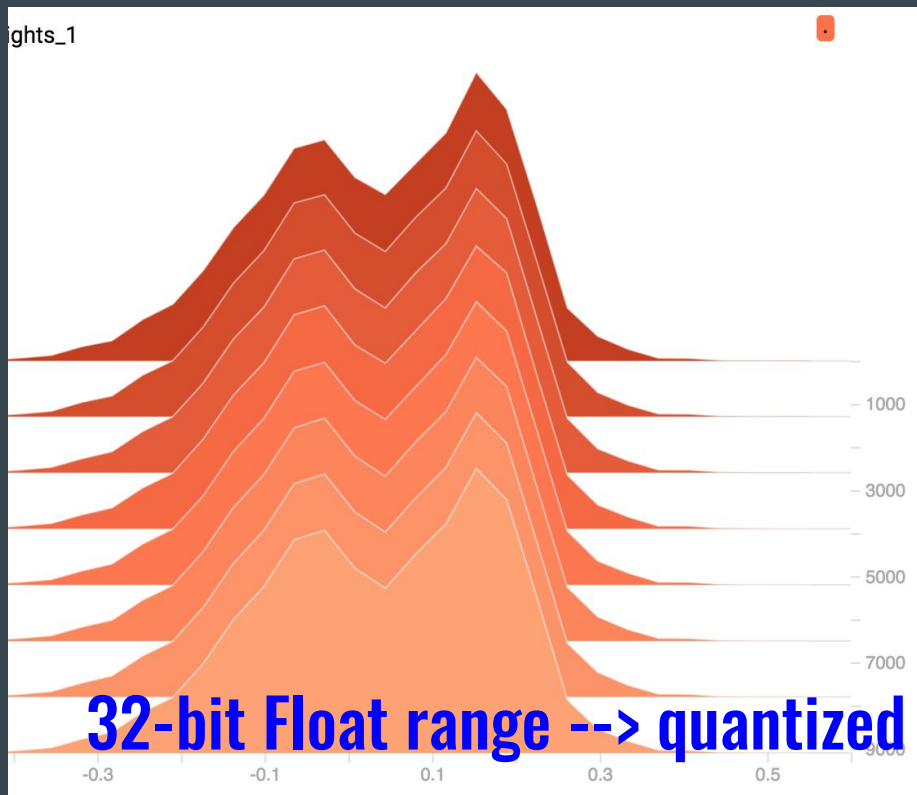
- Train from scratch mIOU 3.4% w/ PASCAL dataset; must need extra COCO augmented dataset

```
eval/miou_1.0_overall[0.0349133685]
```

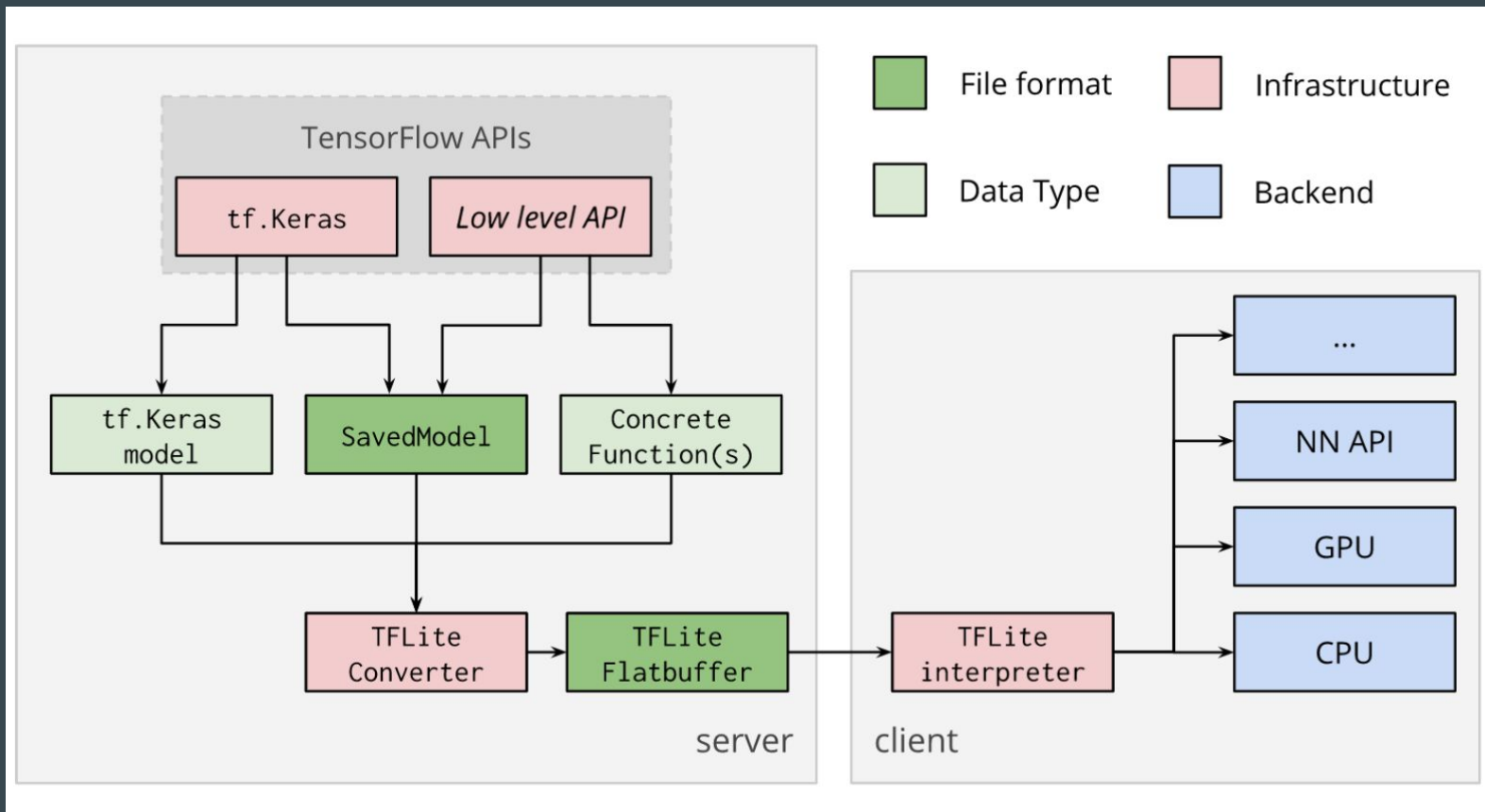
- Overall mIOU improves as increase of iterations of Fine-Tuning FT)

```
FT-10000.eval.out:eval/miou_1.0_overall[0.774983525]  
FT-1000.eval.out:eval/miou_1.0_overall[0.758140802]  
FT-100.eval.out:eval/miou_1.0_overall[0.753919125]  
FT-10.eval.out:eval/miou_1.0_overall[0.75313288]  
FT-1.eval.out:eval/miou_1.0_overall[0.752866758]
```

Logit Weights Distribution from Fine-Tuning (32-bit model)



Quantization with TFLite (chehckpoints vs. SavedModel)



Per-Tensor Quantization {'quantization': scales, zero_points}

```
def _get_tensor_details(self, tensor_index):  
    """Gets tensor details.
```

Args:

tensor_index: Tensor index of tensor to query.

Returns:

A dictionary containing the following fields of the

'name': The tensor name.

'index': The tensor index in the interpreter.

'shape': The shape of the tensor.

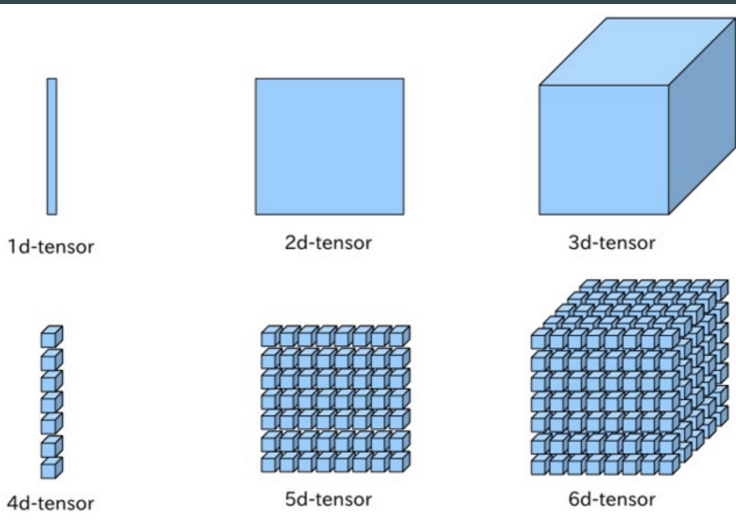
'quantization': Deprecated, use 'quantization_parameters'. This field only works for per-tensor quantization, whereas 'quantization_parameters' works in all cases.

'quantization_parameters': The parameters used to quantize the tensor:

'scales': List of scales (one if per-tensor quantization)

'zero_points': List of zero_points (one if per-tensor quantization)

'quantized_dimension': Specifies the dimension of per-axis quantization, in the case of multiple scales/zero_points.



Post-Quantization {'quantization': (scales, zero_points)=(0.0, 0)}

Post Quantized 8-bit model NO fine-tuning model

```
interpreter=tf.lite.Interpreter(model_path=pascal_post_quantized_8bit_nofinetune)
interpreter.allocate_tensors()
input_details=interpreter.get_input_details()
output_details=interpreter.get_output_details()
h=input_details[0]['shape'][1]
w=input_details[0]['shape'][2]
```

```
print(input_details)
```

```
image = Image.open("test.jpg").resize((h,w))
image = np.array(image)/255.
input_data=np.expand_dims(image, axis=0)
```

```
interpreter.set_tensor(input_details[0]['index'], input_data.astype(np.float32))
interpreter.invoke()
output_data=interpreter.get_tensor(output_details[0]['index'])
results=np.squeeze(output_data)
```

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


Post-Quantization Results

Evaluation on the 16-bit Quantized Model (no-FT) w/r/t 32-bit Model

```
In [57]: import numpy as np
synchro_toFLOAT = np.sum(np.equal(GT_Matrix, noFineTune_16bit_Matrix))/float(GT_Matrix.size)
print(synchro_toFLOAT)

0.574524355072216
```

Comparison between 16-bit Quantized Model vs 8-bit Quantized Model

```
In [60]: synchro_Quants = np.sum(np.equal(noFineTune_16bit_Matrix, noFineTune_8bit_Matrix))/float(noFineTune_16bit_Matrix.size)
print(synchro_Quants)

0.9890906603741322
```

6. Quantization Approach 2 -- Quantization-Aware-Training

=> F32 model prep -> Quantize

Quantization Aware Training : Pretrained Model

Pre Trained
DeepLab V3 Model



Fine Tuning

Without
Quantization

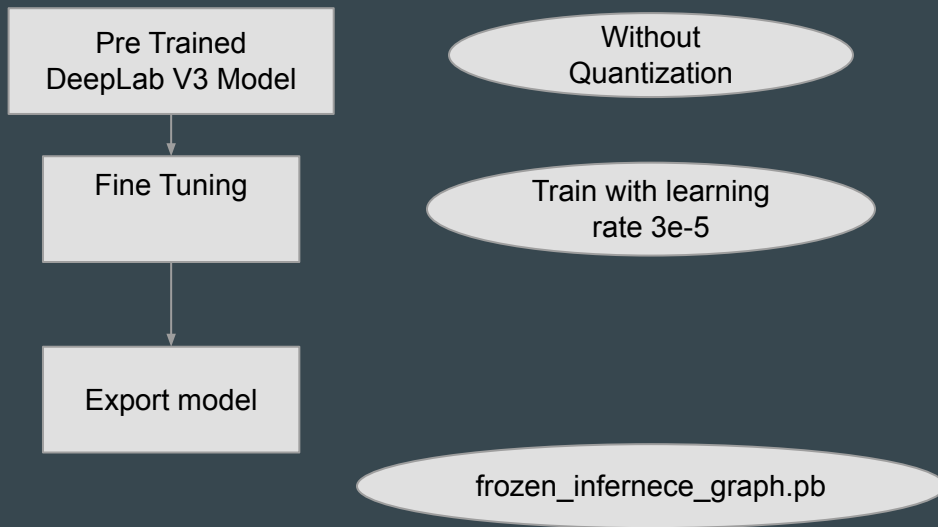
Train with learning
rate $3e-5$

Quantization Aware Training : Pretrained Model



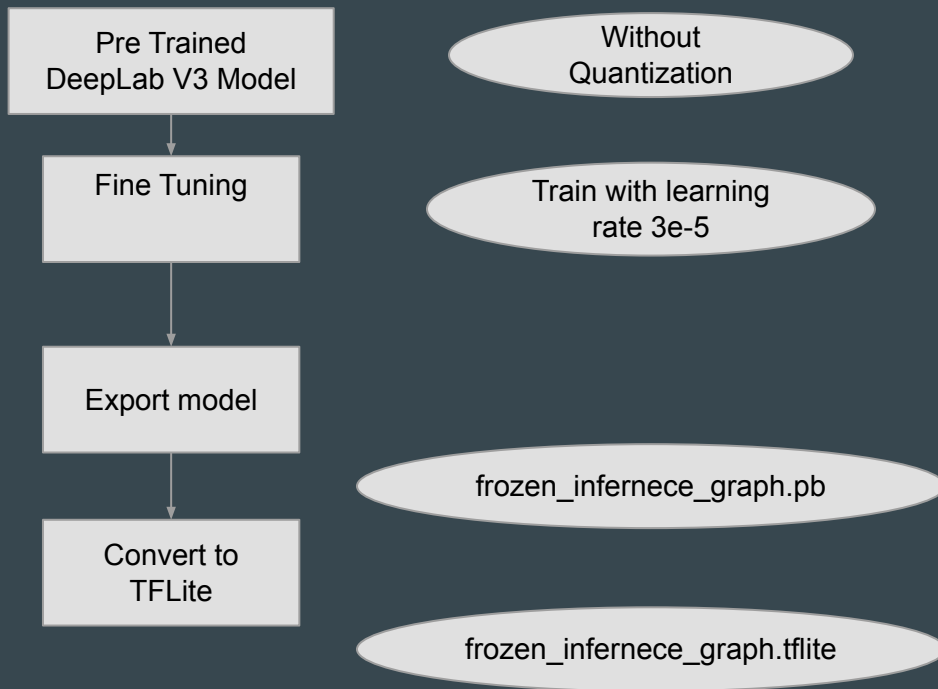
```
python deeplab/train.py  
--base_learning_rate=3e-5  
--training_number_of_steps=3000  
--tf_initial_checkpoint=${PATH_TO_TRAINED_FLOAT_MODEL}
```

Quantization Aware Training : Pretrained Model



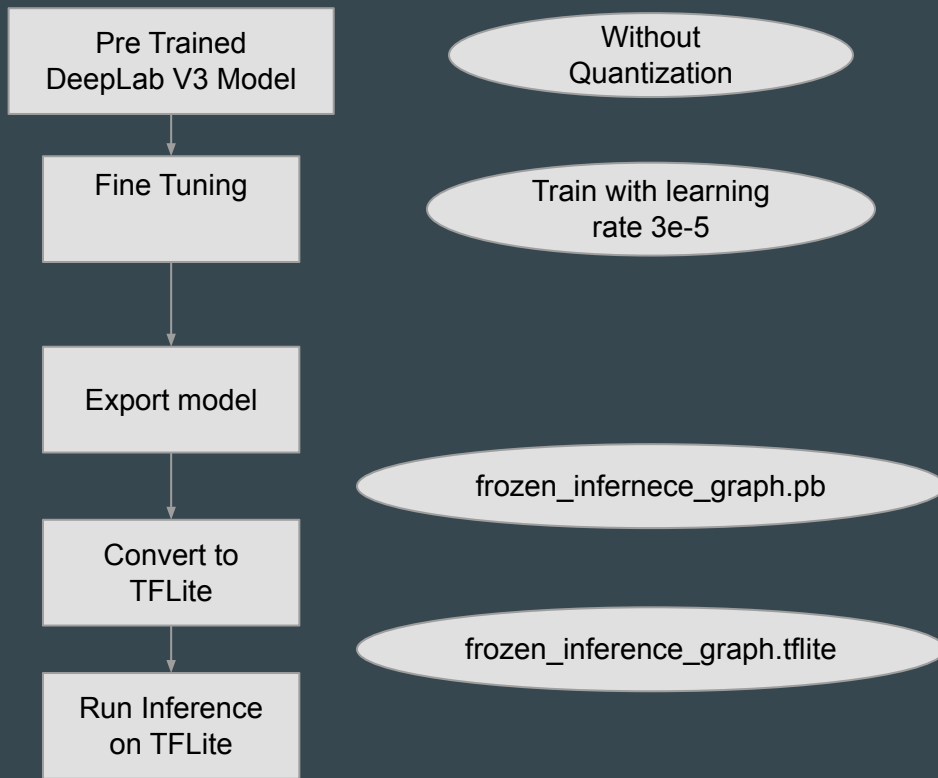
```
# From tensorflow/models/research/  
python deeplab/export_model.py \  
  --checkpoint_path=${CHECKPOINT_PATH} \  
  --quantize_delay_step=0 \  
  --export_path=${OUTPUT_DIR}/frozen_inference_graph.pb
```

Quantization Aware Training : Pretrained Model



```
tflite_convert \  
  --graph_def_file=${OUTPUT_DIR}/frozen_inference_graph.pb \  
  --output_file=${OUTPUT_DIR}/frozen_inference_graph_quant.tflite \  
  --output_format=TFLITE \  
  --inference_type=QUANTIZED_UINT8 \  
  --inference_input_type=QUANTIZED_UINT8 \
```


Quantization Aware Training : Pretrained Model



Quantization Aware Training : Pretrained Model

Model Size:

Non Quantized	Quantized
8.6 Mb	2.1 Mb

How is the Quantization Aware Training Performed?

- Parameter to the training script “`--quantize_delay_step`”
- To create `tf.contrib.create_training_graph()`
- Used to train models for quantization..

Delayed Quantization

- One of the .Next items that we showcased for the final milestone
- Used when training and quantization have to be performed in the same cycle.
- Quantization will be delayed until the last few steps.
- Controlled by the argument 'quant_delay' for the function `tf.contrib.create_training_graph()`

Delayed Quantization

- Trained a DeeplapV3+ model from scratch with the following parameters

```
python train.py
```

```
--training_number_of_steps=36000
```

```
--quantize_delay_step=30000
```

```
--base_learning_rate=0.0001
```

7. Inference and Evaluation with Demo (Quantization/Inference)

=> Comparison between F32 vs. 8-bit integer inferences

Demo on the Colab by Searching our GitHub

<https://github.com/SherifSabri/ImageSegmentationWithDeeplab>
CS256-GroupE_inference_deeplab.ipynb

Examples

Recent

Google Drive

GitHub

Upload

Enter a GitHub URL or search by organization or user


☐ Include private repos



<https://github.com/SherifSabri/ImageSegmentationWithDeeplab.git>


Repository: [🔗](#)
SherifSabri/ImageSegmentationWithDeeplab ⌵



Branch: [🔗](#)
master ⌵


Path



 .ipynb_checkpoints/inference_deeplab-checkpoint.ipynb

 CS256_GroupE_PostQuantization.ipynb

 CS256_GroupE_inference_deeplab.ipynb

←inference evaluation

Post-Training Quantization Script is available

<https://github.com/SherifSabri/ImageSegmentationWithDeeplab>
CS256-GroupE_PostQuantization.ipynb

Examples

Recent

Google Drive


GitHub


Upload

Enter a GitHub URL or search by organization or user


☐ Include private repos



<https://github.com/SherifSabri/ImageSegmentationWithDeeplab.git>


Repository:  SherifSabri/ImageSegmentationWithDeeplab



Branch:  master


Path



 .ipynb_checkpoints/inference_deeplab-checkpoint.ipynb


 



 CS256_GroupE_PostQuantization.ipynb

 CS256_GroupE_inference_deeplab.ipynb

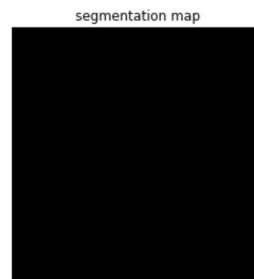
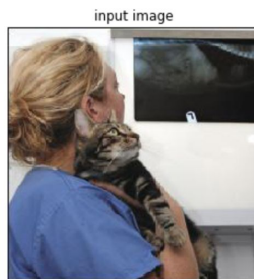
 inference_deeplab.ipynb

← Quantization Script

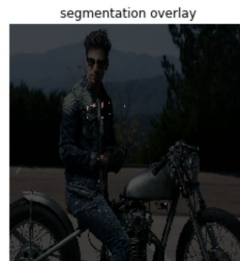
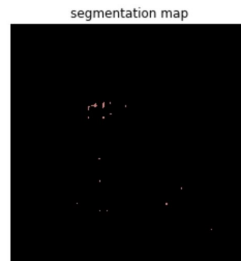
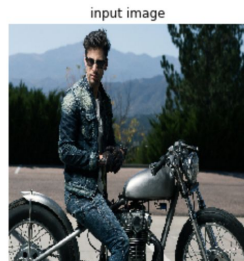
Initial Results

Failed to produce segmentation mask, and an invalid quantization field



Initial Results

Still Failed to produce
segmentation mask, with a valid
quantization field

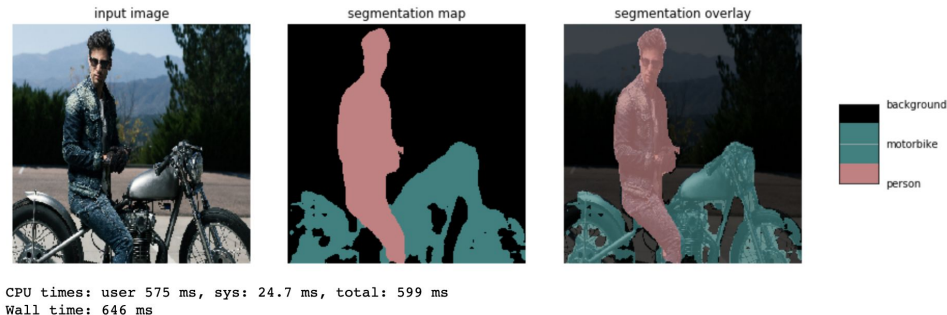


background
person

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

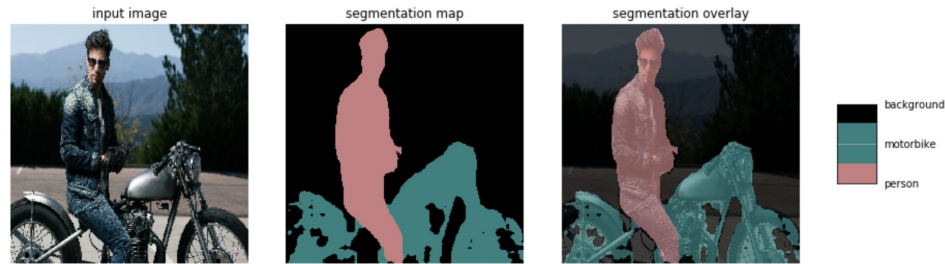
Final Results

Successful inference on quantized
model

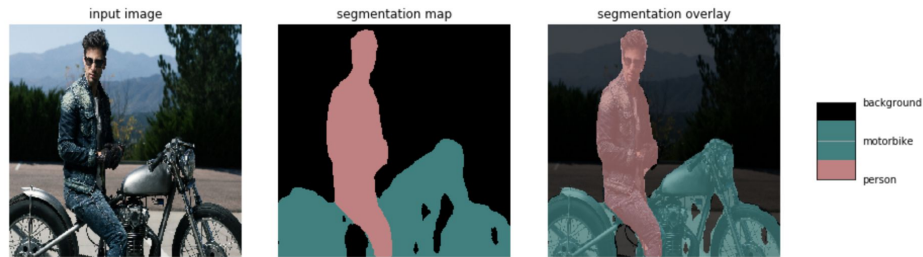


Comparison with Deeplab authors provided model

Comparable results both in
accuracy and inference time



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



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

Milestone 2.5 Slides

Introduction

- Machine Learning at the Edge
 - Increasing need for running the prediction and ML models at various edge devices
 - Nature of the edge devices provides a constraint on both compute and memory
 - Large Models are harder to distribute to edge devices
- Neural Networks are getting larger and more complex
 - Training ML models is a time and compute intensive task
 - Training ML models is inefficient in terms of energy consumption and overall cost

Methods for Improving Efficiency of Deep Learning

- Improve the Hardware doing the training and running the models
 - Faster CPUs & GPUs
 - Use specialized hardware for machine learning (Ex: FPGAs)
- Improve the efficiency of the algorithms responsible for training and predicting the machine learning models
 - Use Model compression techniques

Model Compression Techniques

- Quantization
- Weight sharing
- Pruning
- Architecture design
- Knowledge Distillation
- Others (low rank approximation, winograd transformation, ...etc)

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.

Networks Choice

We choose to carry out our research using “MobileNet v2” as the backbone network due to its excellent performance and benchmarks on a variety of tasks and the small size and lower latency of its produced models, For the decoder network, we choose to use DeepLab v3+ that’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.

Datasets

- Pascal VOC dataset -the main dataset for the majority of the research-
- Cityscapes dataset -experimented with it for quantization aware training and inference-
- ADE20K dataset -used briefly for running inference-

Details

- VOC 2012 dataset
- CityScapes dataset
- DeepLab V3+ Decoder Network
- TensorFlow 1.14
- TensorFlow lite

TensorFlow Lite 8-bit Quantization

- 8-bit quantization approximates floating point values using the following formula
- $\text{Real_value} = (\text{int8_value} - \text{zero_point}) \times \text{scale}$
- Activations are asymmetric: they can have their zero-point anywhere within the signed int8 range $[-128, 127]$
- Weights are symmetric: forced to have zero-point equal to 0.

Quantization Aware Training

- Quantization aware Training is very powerful method reducing model size and Improving inference speed
- Most Recent techniques are Binarization of the Neural Network
- Some of the examples are
 - Binary Connect
 - BinaryNet
 - Binary Weight Network

Binary-Connect

- W = weights with full-precision
- W_b weight with quantized weight
- Only weights are binarized
- $\sigma(w)$ hard sigmoid
- Weight Binarized for forward-propagation and backward-propagation but not for parameter update

Deterministic vs stochastic binarization

➔ Deterministic:

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

➔ Stochastic

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

$$\sigma(w) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right)$$

Binary Connect Algorithm

Algorithm 1 SGD training with BinaryConnect. C is the cost function for minibatch and the functions $\text{binarize}(w)$ and $\text{clip}(w)$ specify how to binarize and clip weights. L is the number of layers.

Require: a minibatch of (inputs, targets), previous parameters w_{t-1} (weights) and b_{t-1} (biases), and learning rate η .

Ensure: updated parameters w_t and b_t .

1. Forward propagation:

$w_b \leftarrow \text{binarize}(w_{t-1})$

For $k = 1$ to L , compute a_k knowing a_{k-1} , w_b and b_{t-1}

2. Backward propagation:

Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$

For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and w_b

3. Parameter update:

Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{\partial b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}

$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$

$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

Binary Neural Network

Binary Connect with activations binarized too.

Require: a minibatch of inputs and targets (a_0, a^*) , previous weights W , previous BatchNorm parameters θ , weights initialization coefficients from (Glorot & Bengio, 2010) γ , and previous learning rate η .

Ensure: updated weights W^{t+1} , updated BatchNorm parameters θ^{t+1} and updated learning rate η^{t+1} .

{1. Computing the parameters gradients:}

{1.1. Forward propagation:}

for $k = 1$ to L **do**

$W_k^b \leftarrow \text{Binarize}(W_k)$

$s_k \leftarrow a_{k-1}^b W_k^b$

$a_k \leftarrow \text{BatchNorm}(s_k, \theta_k)$

if $k < L$ **then**

$a_k^b \leftarrow \text{Binarize}(a_k)$

end if

end for

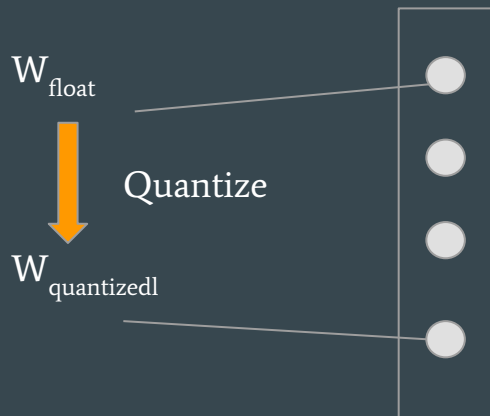
Binary Weight Network

- This network uses same concepts as BinaryConnect
- The way weights are binarized in different way.
- \mathbf{W} - real Valued Weights
- \mathbf{I} - real valued input tensor
- $*$ - Convolution operation
 - $\mathbf{I} * \mathbf{W} \sim (\mathbf{I} \pm \mathbf{B})\alpha$
- \mathbf{B} - Binary Weights (sign of \mathbf{W})
- \pm Convolution with only addition/Subtraction
- α - Real valued scalar factor (Average of \mathbf{W})

$$\alpha^* = \frac{\mathbf{W}^\top \text{sign}(\mathbf{W})}{n} = \frac{\sum |\mathbf{W}_i|}{n} = \frac{1}{n} \|\mathbf{W}\|_{\ell_1}$$

Our Approach

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



Results on other Architecture

Other Architectures

	ResNet-18		GoogLenet	
Network Variations	top-1	top-5	top-1	top-5
Binary-Weight-Network	60.8	83.0	65.5	86.1
XNOR-Network	51.2	73.2	N/A	N/A
Full-Precision-Network	69.3	89.2	71.3	90.0