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

Overview

- **Computer Vision Topic -- Semantic Segmentation**
  - Hard problem: object detection + multi-classification at pixel level
- **Segmentation Framework -- DeepLabV3+**
  - Atrous convolution (faster)
  - Encoder-Decoder (shaper boundary)
- **Backbone Architecture -- MobileNetV2**
  - Depthwise-separable-convolution + Inverted-Residual-Layer
  - TFLite for Mobile & IoT
- **Project Task -- Quantization**
  - Quantization makes DeepLabV3+ on MobileNetV2 more lighter and faster

Overview

- **Quantization Approach 1 -- Post-Training-Quantization**
  - F32 model prep -> Quantize
- **Quantization Approach 2 -- Quantization-Aware-Training**
  - F32 model prep -> Quantize
- **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

- Understanding the image at pixel level
- Partitioning the image into regions of meaningful objects
- Assign a category label to each object

Applications

- Self driving cars
- Medical imaging
- Photography - spoilers ! -

Semantic Segmentation Requires

- pinpointing the outline of objects
- Imposing much stricter localization accuracy than image-level classification
- Conv layers with downsampling and upsampling

PASCAL VOC 2012 Dataset for Semantic Segmentation

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

background
aeroplane
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
...

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

Trends

- Model sizes getting larger
- Computationally expensive
- Energy in-efficient

Dally, NIPS'2016 workshop on Efficient Methods for Deep Neural Networks

Challenges

- Distribution
- Inference time
- Memory restriction

Hard to distribute large models through over-the-air update

App icon is in the public domain
Phone image is licensed under CC-BY 2.0

This image is licensed under CC-BY 2.0

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)

Quantization

- Convert Higher precision to Lower precision
- Example 256 bits per pixel to 2 bits per pixel

Weight Sharing

- Non-linear quantization
- Represent weights as their approximate integer values (2.03, 1.89 → 2)
- Share the same weights using cluster index

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
- MobileNet v2 Encoder Network
- TensorFlow 1.15
- TensorFlow lite
- p2.xLarge ec2 instance
- Deep Learning AMI with Ubuntu 16

2. Segmentation Framework -- DeepLabV3+

=> Atrous convolution (faster)

=> Encoder-Decoder (shaper feature extraction)

=> Spatial Pyramid Pooling

Background

- Shallow depth of field effect
- Separating foreground (person) from background
- Example: portrait mode in Pixel phones

History

- Close to 4 years of development
- Only recently open sourced (2016)
- Currently at version deeplab v3+
- Particularly useful for semantic segmentation

Available backbone networks

- MobileNetv2 and MobileNetv3 : A fast network structure designed for mobile devices.
- Xception : A powerful network structure intended for server-side deployment.
- ResNet-v1-{50,101} : We provide both the original ResNet-v1 and its 'beta' variant where the 'stem' is modified for semantic segmentation.
- PNASNet : A Powerful network structure found by neural architecture search.
- Auto-DeepLab : A segmentation-specific network backbone found by neural architecture search.

Architecture

Encoder

- Uses a pre-trained CNN to extract the essential information
- Depthwise separable convolution instead of max pooling
- Extra batch normalization and Relu

Decoder

- Reconstrct output (segmentation mask)
- Uses ASPP
- Bilinear upsampling to get the correct spatial dimension

Main Challenge: Objects at multiple scales (sizes)

(a) Image Pyramid (b) Encoder-Decoder (c) Deeper w. Atrous Convolution (d) Spatial Pyramid Pooling

Figure 2. Alternative architectures to capture multi-scale context.

Spatial pyramid pooling

- Capture context at several ranges
  - I.e: multiple scaled versions of the input
- Parallel atrous convolution layers with different rates capture multi-scale information
  - I.e: parallel versions of the same network training at different scales, then combined into one feature
- atrous spatial pyramid pooling (ASPP)
  - Efficient computational complexity
  - Larger field of view using same number of parameters as traditional convolution

Traditional Convolution

2D convolution using a kernel size of 3, stride of 1 and padding of 1

Atrous Convolution

(Dilated Convolutions)

2D convolution using a kernel size of 3, stride of 1, no padding, and an extra dilation rate of 2

Faster "Atrous Depthwise Separable" Conv. on MobileNetV2

- Depthwise Convolution -> fewer parameters
- Pointwise Convolution -> more channels

Sharper Boundary Segmentation by Encoder-Decoder

- Extra decoder module to refine the segmentation results especially along object boundaries
- Applying depthwise separable convolution to both ASPP and decoder modules, resulting in a faster and stronger encoder-decoder network for semantic segmentation

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

Encoder: DeepLabV3 feature map

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

MobileNetV2:

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

MobileNetV2

Residual block layer

1x1 conv2d , ReLU6
3x3 dwise s=s, ReLU6
linear 1x1 conv2d

All layers

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

MobileNetV2 using TensorFlow Lite for Mobile & IoT

TensorFlow Instal

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

Build TensorFlow Lite

Build for iOS

Build for ARM64

Build for Raspberry Pi

4. Project Task -- Quantization

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

Fixed Point, Per-Tensor Quantization 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, `[min_fixed, max_fixed].`
4. With input tensor `T`, `num_bits = 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 Quantization implemented in TensorFlow

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

q = (r/S) + z

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 Quantization implemented in TensorFlow

equivalent converted subgraph, still with float inputs and outputs, but with internal conversions so the calculations are done in eight bit.

Quantization Experiments

Overview of Our Quantization Evaluation

A
Reference 32-float Model Inference
CPU times: user 2.25 s, sys: 184 ms, total: 2.43 s
Wall time: 2.46 s

B
Original Quantized Model
CPU times: user 384 ms, sys: 6.42 ms, total: 390 ms
Wall time: 397 ms

C
Successful Inference on Quantize Aware Model (QAT)
CPU times: user 402 ms, sys: 8.75 ms, total: 411 ms
Wall time: 417 ms
Evaluation on QAT-8bit w/r/t 32-bit float model
0.935706713176704

D
Post Quantized 8-bit model NO fine-tuning model
CPU times: user 418 ms, sys: 4.81 ms, total: 422 ms
Wall time: 427 ms
Evaluation on the 8-bit Quantized Model (no-FT) w/r/t 32-bit Model
0.9831363116476485

Post Quantized 16-bit model NO fine-tuning model
CPU times: user 417 ms, sys: 11.3 ms, total: 428 ms Wall time: 440 ms
Evaluation on the 16-bit Quantized Model (no-FT) w/r/t 32-bit Model¶
0.9866397638019676

E
Post Quantized 8-bit model 10K iterations fine-tuning model
CPU times: user 411 ms, sys: 15.7 ms, total: 427 ms
Wall time: 436 ms
Evaluation on the 8-bit Quantized Model (FT-10K) w/r/t 32-bit Model
0.9685905254798247

Post Quantized 16-bit model 10K iterations fine-tuning model
CPU times: user 421 ms, sys: 16.9 ms, total: 438 ms
Wall time: 441 ms
Evaluation on the 16-bit Quantized Model (FT-10K) w/r/t 32-bit Model
0.9758786179223237

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[0.753266752]
```

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

Quantization Models / Script

https://github.com/SherifSabri/ImageSegmentationWithDeeplab

ImageSegmentationWithDeeplab / models /

Ubuntu fine-tuning mo

..

Evals

FT-1

FT-10

FT-100

FT-1000

FT-10000

FT-NO

GTmodel/deeplabv3_mnv2_pascal_train_aug

ImageSegmentationWithDeeplab / models / research / deeplab / postQuantizationFineTuning

Ubuntu fine-tuning models for post quantization

..

events.out.tfevents.1575568066.ip-172-31-60-... fine-tuning models for post quantization

frozen_inference_graph.pb for post quantization

postQuant_FT10000_FLOAT.tflite for post quantization

postQuant_FT10000_FLOAT_minmax.tflite for post quantization

postQuant_FT10000_float16_minmax.tflite for post quantization

Full 32-bit

Quantized-8bit

Quantized-16bit

Tensorflow Graph prior to Quantization (float32)

Quantized Graph (16-bit int) : Almost same size as f32; but operations are pruned

NODE PROPERTIES

type	Conv2D
name	94

ATTRIBUTES

padding	SAME
stride_w	1
stride_h	1
fused_activation...	RELU6
dilation_w_factor	1
dilation_h_factor	1

INPUTS last 15th layer

input	id: **MobilenetV2/expanded_conv_15/add**
weights	id: **MobilenetV2/expanded_conv_16/expand/weights/read**
bias	id: **MobilenetV2/expanded_conv_16/expand/Conv2D_bias**

OUTPUTS

output	id: **MobilenetV2/expanded_conv_16/expand/Relu6**

Quantized Graph (8-bit int) : ¼ size reduction; dequantization for intermediate operations

NODE PROPERTIES

type	Conv2D
name	108

ATTRIBUTES

padding	SAME
stride_w	1
stride_h	1
fused_activation...	RELU6
dilation_w_factor	1
dilation_h_factor	1

INPUTS

last 15th layer

input	id: **MobilenetV2/expanded_conv_15/add**
weights	id: **MobilenetV2/expanded_conv_16/expand/weights/read**
bias	id: **MobilenetV2/expanded_conv_16/expand/Conv2D_bias**

OUTPUTS

output	id: **MobilenetV2/expanded_conv_16/expand/Relu6**

```
CPU times: user 409 ms, sys: 11.2 ms, total: 421 ms
Wall time: 426 ms
```

Evaluation on the 8-bit Quantized Model (no-FT) w/r/t 32-bit Model

```python
import numpy as np
synchro_toFLOAT = np.sum(np.equal(GT_Matrix, noFineTune_8bit_Matrix))/float(GT_Matrix.size)
print(synchro_toFLOAT)
```

```
0.9844738552033104
```

Evaluation Ref: 32-bit inference

8-bit int Quantized

Size: 8.4M→ 2.1M
IOU: 98% similar
Latency: 2.46s → 427ms (5.8 faster)

Evaluation on Post Quantization Models w/r/t float32 Model

Model	Size (MB)	Latency (ms) Walltime	IOU w/r/t 32-bit Inference
32-bit float	8.4	2460	1.0
8-bit int (no-FT)	2.1 **(1/4)**	427 **(x 5.76 faster)**	0.9844
16-bit int (no-FT)	8.1	440 **(x 5.59 faster)**	0.9875
8-bit int (10K-FT)	2.1 **(1/4)**	436 **(x 5.64 faster)**	0.9714
16-bit int (10K-FT)	8.1	441 **(x 5.58 faster)**	0.9779

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

=> F32 model prep -> Quantize

Why ?

- Float 32 Model - Forward pass - training / inference - full float precision computation
- Quantized Mode - Forward pass - inference - integer precision
- Loss in accuracy
- In a quantized model, no way to make up for this loss.

Quantization Aware Training (QAT)

- Simulate quantization during training
  - Perform computations on rounded weight values

Quantization Aware Training (QAT)

- Simulate quantization during training
  - Perform rounded computations
- Fake quantization nodes
  - Capture the activation range of each layer
  - Perform quantized computations

Note: Weights are maintained in full precision.

Quantization Aware Training (QAT)

- Simulate quantization during training
  - Perform rounded computations
- Fake quantization nodes
  - Capture the activation range of each layer
  - Perform quantized computations

- Prepares model weights for the loss incurred by quantization.

Adding QAT Support - Tensorflow

- Modification 1:

Create training Graph:

```
tf.contrib.quantize.create_training_graph()
```

argument : **`quant_delay`**

Rewrites the training graph in place by adding fake quantization nodes.

Adding QAT Support - Tensorflow

- Modification 2:

Create Evaluation Graph:

```
tf.contrib.quantize.create_eval_graph()
```

Rewrites the evaluation graph in place by adding fake quantization nodes.

QAT - Experiments

Graph Visualization

Tensorflow Graph prior to Quantization (float32)

Graph Visualization Tensorflow Graph with QAT

Graph Visualization Tensorflow Graph with QAT

Graph Visualization TFLite Graph With Quantization

dilation_h_factor	1

INPUTS

input	id: **MobilenetV2/MobilenetV2/input**
weights	**lobilenetV2/Conv/weights_quant/FakeQuantWithMinMaxVars**
bias	id: **MobilenetV2/Conv/Conv2D_Fold_bias**

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

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

Post-Training Quantization Script is available

https://github.com/SherifSabri/ImageSegmentationWithDeeplab

CS256-GroupE_PostQuantization.ipynb

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
```

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

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

Final Results

Successful inference on quantized model

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