TF-TRT BEST PRACTICE, EAST AS AN EXAMPLE

Xiaowei Wang (王晓伟), Dec 18th, 2019
OUTLINE

- Background
- TFTRT
- TRT API
- TRT UFF Parser
- Conclusion
A fully-convolutional network (FCN) adapted for text detection that outputs dense per-pixel predictions of words or text lines.

https://arxiv.org/abs/1704.03155
Use the **ResNet-50** as the backbone instead.

Each block contains several units.

https://github.com/argman/EAST
TRT Acceleration

- TRT UFF Parser: Parse the network from the TF model
- TRT API: Create the network from scratch
- TF TRT: Convert the TF graph to the TRT graph directly
TFTRT (TensorFlow integration with TensorRT) parses the frozen TF graph and converts each supported subgraph to a TRT optimized node (TRTEngineOp), allowing TF to execute the remaining graph.

Create a frozen graph from a trained TF model, and give it to the Python API of TF-TRT.

SETUP

Install:

TFTRT is part of the TensorFlow binary, which means when you install tensorflow-gpu, you will be able to use TF-TRT too. (pip install tensorflow-gpu)

prerequisite:

```python
import tensorflow as tf
import tensorflow.contrib.tensorrt as trt

inputs = "input_images"
outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]

model_infer.ckpt-49491.data-00000-of-00001
model_infer.ckpt-49491.index
model_infer.ckpt-49491.meta
```
Step 1  Obtain the TF frozen graph

• With Ckpt

```python
with tf.Session( ) as sess:
    # Import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]  # node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def, output_node_names=outputs)
```

• With Pb

```python
with tf.Session( ) as sess:
    # deserialize the frozen graph
    with tf.gfile.Gfile("./model.pb", "rb") as f:
        frozen_graph = tf.GraphDef()
        frozen_graph.ParseFromString(f.read())
```

Step 2  Create the TRT graph from the TF frozen graph

```python
trt_graph = trt.create_inference_graph (  
    input_graph_def = frozen_graph, outputs = output_node_name,
    max_batch_size = 1, max_workspace_size_bytes = 1<<30,
    precision_mode = ="FP32",
    minimum_segment_size = 5, ... )
```

- **input_graph_def**: the frozen TF GraphDef object
- **outputs**: the names list of output nodes
- **max_batch_size**: maximum batch size
- **max_workspace_size_bytes**: maximum GPU memory size available for TRT layers
- **precision_mode**: FP32 / FP16 / INT8
- **minimum_segment_size**: determine the minimum number of nodes in a TF sub-graph for the TRT engine to be created

Step 3  Import the TRT graph and run

```python
# import the TRT graph into the current default compute graph
g = tf.get_default_graph()
inputs = g.get_tensor_by_name("input_images:0")
outputs = [n + ':0' for n in outputs]  # tensor names
f_score, f_geo = tf.import_graph_def(trt_graph, input_map={"input_images": inputs},
                                                 return_elements=outputs, name="")

# run the optimized graph in session
img = cv2.imread("xxx.jpg")
score, geometry = sess.run([f_score, f_geo], feed_dict={inputs: [img]})
```

TFTRT FP32

```python
with tf.Session() as sess:
    # create a `Saver` object, import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ['"feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"'] # node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def, output_node_names=outputs)

    # create a TRT inference graph from the TF frozen graph
    trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
                                           max_batch_size=1, max_workspace_size_bytes=1<<30,
                                           precision_mode="FP32",
                                           minimum_segment_size=5)

    # import the TRT graph into the current default graph
    g = tf.get_default_graph()
    input_images = g.get_tensor_by_name("input_images:0")
    outputs = [n+':0' for n in outputs] # tensor names
    f_score, f_geometry = tf.import_graph_def(trt_graph, input_map="input_images":input_images,
                                              return_elements=outputs, name="")

    # run the optimized graph in session
    img = cv2.imread("./img.jpg")
    score, geometry = sess.run([f_score, f_geometry], feed_dict={input_images: [img]})
```

with tf.Session() as sess:
    # create a `Saver` object, import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]  # node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def, 
        output_node_names=outputs)

    # create a TRT inference graph from the TF frozen graph
    trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs, 
        max_batch_size=1, max_workspace_size_bytes=1<<30, 
        precision_mode="FP16", 
        minimum_segment_size=5)

    # import the TRT graph into the current default graph
    g = tf.get_default_graph()
    input_images = g.get_tensor_by_name("input_images:0")
    outputs = [n + ':0' for n in outputs]  # tensor names
    f_score, f_geometry = tf.import_graph_def(trt_graph, input_map="input_images":input_images, 
        return_elements=outputs, name="")

    # run the optimized graph in session
    img = cv2.imread("./img.jpg")
    score, geometry = sess.run([f_score, f_geometry], feed_dict={input_images: [img]})

TFTRT converts the native TF subgraph (TRTEngineOp_0_native_segment) to a single TRT node (TRTEngineOp_0).
TFTRT INT8

The INT8 precision mode requires an additional **calibration step** before quantization.

**Calibration**: run inference in FP32 precision on a calibration dataset, which collects required statistics and runs the calibration algorithm, to generate INT8 quantization (scaling factors) of the weights and activations in the trained TF graph.

\[
\text{INT8\_value} = \text{FP32\_value} \times \text{scale}
\]

TFTRT INT8

Step 1  Obtain the TF frozen graph (trained in FP32)

Step 2  Create the calibration graph -> Execute it with calibration data -> Convert it to the INT8 optimized graph

# create a TRT inference graph, the output is a frozen graph ready for calibration
```
calib_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
                                          max_batch_size=1, max_workspace_size_bytes=1<<30,
                                          precision_mode="INT8", minimum_segment_size=5)
```

# Run calibration (inference) in FP32 on calibration data (no conversion)
```
f_score, f_geo = tf.import_graph_def(calib_graph, input_map={"input_images":inputs},
                                         return_elements=outputs, name="")
```

Loop img: score, geometry = sess.run([f_score, f_geo], feed_dict={inputs: [img]})

# apply TRT optimizations to the calibration graph, replace each TF subgraph with a TRT node optimized for INT8
```
trt_graph = trt.calib_graph_to_infer_graph(calib_graph)
```

Step 3  Import the TRT graph and run

...
TFTRT FP32/FP16/INT8 Performance  (V100, batch size = 1)

<table>
<thead>
<tr>
<th>ICDAR2015 TestSet (672x1280)</th>
<th>FPS</th>
<th>recall</th>
<th>precision</th>
<th>F1score</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF Slim</td>
<td>42</td>
<td>0.7732</td>
<td>0.8466</td>
<td>0.8083</td>
</tr>
<tr>
<td>TFTRT FP32</td>
<td>63</td>
<td>0.7732</td>
<td>0.8466</td>
<td>0.8083</td>
</tr>
<tr>
<td>TFTRT FP16</td>
<td>98</td>
<td>0.7723</td>
<td>0.8442</td>
<td>0.8066</td>
</tr>
<tr>
<td>TFTRT INT8</td>
<td>83</td>
<td>0.7602</td>
<td>0.8572</td>
<td>0.8058</td>
</tr>
</tbody>
</table>

INT8 with IDP.4A instruction is slower than FP16 with Tensor Core on V100.

FP16

- 1.3% void Eigen::internal::EigenMetaKernel<Eigen::TensorEvaluator<Eigen:
- 1.1% trt volta h884cudnn_256x128 ldg8 relu_exp_small_nhwc tn v1
- 1.0% void tensorflow::functor::RowReduceKernel<float* tensorflow::Tensor:

INT8

- 0.1% void cuPad::pad<char, int, int=128, bool=1>(int*, int, cuPad::p:
- 0.1% void trt volta fp32 icudnn int8x4 128x128 relu interior nn v1
- 0.1% void fused:fusedConvolutionReluKernel<fused::SrcChw

h884cudnn: HMMA for Volta, fp16 input, output, and accumulator.

fp32_icudnn_int8x4: Int8 kernels using the IDP.4A instruction. Inputs are aligned to fetch 4x int8 in one instruction.

https://docs.google.com/spreadsheets/d/1xAo6TcSgHdd25EdQ-6GqM0VKBTYu8cWyycgJhHRVlgY/edit#gid=1454841244
TAKEAWAYS

- The names of input and output nodes
  
  ```
  inputs = "input_images"
  outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]
  ```

- The TF model trained in FP32 (checkpoint or pb files)
  
  ```
  model_infer.ckpt-49491.data-00000-of-00001
  model_infer.ckpt-49491.index
  model_infer.ckpt-49491.meta
  ```

- Calibration dataset for INT8 quantization
  
  ```
  img_113.jpg  img_159.jpg  img_203.jpg  img_249.jpg
  img_114.jpg  img_15.jpg   img_204.jpg  img_24.jpg
  img_115.jpg  img_160.jpg  img_205.jpg  img_250.jpg
  ```
Tips 1: GPU memory allocation

Specify the fraction of GPU memory allowed for TF, making the remaining available for TRT engines. Use the `per_process_gpu_memory_fraction` and `max_workspace_size_bytes` parameters together for best overall application performance.

```python
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.6)
tf_config = tf.ConfigProto(gpu_options=gpu_options, allow_soft_placement=True)
with tf.Session(config=tf_config) as sess:

trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
                                       max_batch_size=1,
                                       max_workspace_size_bytes=1<<30,
                                       precision_mode="FP32",
                                       minimum_segment_size=5)
```

Certain algorithms in TRT need a larger workspace, therefore, decreasing the TF-TRT workspace size might result in not running the fastest TRT algorithms possible.
Tips 2: Minimum segment size

To achieve the best performance, different possible values of minimum_segment_size can be tested.

We can start by setting it to a large number and decrease this number until the converter crashes.

<table>
<thead>
<tr>
<th>min_seg_size</th>
<th>TRT nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1 (446 tf nodes)</td>
</tr>
<tr>
<td>30</td>
<td>2 (44 / 446 tf nodes)</td>
</tr>
<tr>
<td>5</td>
<td>4 (24 / 24 / 44 / 446 tf nodes)</td>
</tr>
<tr>
<td>3</td>
<td>5 (4 / 24 / 24 / 44 / 446 tf nodes)</td>
</tr>
</tbody>
</table>

`trt.create_inference_graph (...)`, `minimum_segment_size = 5, ...`

determine the minimum number of nodes in a TF sub-graph for the TRT engine to be created
Tips 3: Batch normalization

The FusedBatchNorm operator is converted to TRT only if is_training=False, which indicates whether the operation is for training or inference.

```
(Pdb) frozen_graph.node[176]
name: "model_0/resnet_v1_50/block1/unit_2/bottleneck_v1/conv1/BatchNorm/cond/FusedBatchNorm"
attr {
  key: "is_training"
  value {
    b: true
  }
}
```

tf.train.import_meta_graph(“xxx.ckpt”) just imports the saved graph, usually training graph.

Need to change the is_training=False in the graph.

Tips 3: Batch normalization

Workarounds:

1. With the codes building the network:

   Build the TF inference graph by setting `is_training=False` for all fusedBatchNorm layers, and then restore the weights from the training graph without using `tf.train.import_meta_graph`.

```python
input_images = tf.placeholder(tf.float32, shape=[None, 672, 1280, 3], name='input_images')
f_score, f_geometry = model.model(input_images, is_training=False)
saver = tf.train.Saver()
with tf.Session(config=tf_config) as sess:
    saver.restore(sess, './model.ckpt')
    tf.graph_util.convert_variables_to_constants(sess, tf.get_default_graph(), 'output_graph_def
trt.create_inference_graph...

with slim.arg_scope([slim.batch_norm], is_training=is_training):
```

Tips 3: Batch normalization

Workarounds:

2. Without the codes building the network:

   Resave an inference graph as the ckpt files and then use the `tf.train.import_meta_graph` API directly.

Customer provided:

```python
input_images = tf.placeholder(tf.float32, shape=[None, 672, 1280, 3], name='input_images')
f_score, f_geometry = model.model(input_images, is_training=False)
saver = tf.train.Saver()
with tf.Session(config=tf_config) as sess:
saver.restore(sess, '../model.ckpt')
saver.save(sess, './model_infer.ckpt')
```

Then

```python
saver = tf.train.import_meta_graph("model_infer.ckpt.meta")
saver.restore(sess, "model_infer.ckpt")
```

Tips 4: TRT node name

```python
...  
  tf.import_graph_def(trt_graph, ...)
  g = tf.get_default_graph()
  f_score = g.get_tensor_by_name("Conv_7/Sigmoid_1:0")
...  
  score = sess.run([f_score], feed_dict={inputs: [img]})
```

if the same name has been previously used in the same scope, it will be made unique by appending \_N to it.

https://www.tensorflow.org/api_docs/python/tf/variable_scope
TRT API

Install the TensorRT SDK

import tensorrt as trt

- Load all weights from the saved model (the TF model trained in FP32)
- Create the network from scratch with TRT layer APIs (the network structure)
- Build the TRT engine
- Create the context and execute inference

https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/python_api/index.html
TAKEAWAYS

- The TF model trained in FP32 (checkpoint or pb files)

```
model_infer.ckpt-49491.data-00000-of-00001
model_infer.ckpt-49491.index
model_infer.ckpt-49491.meta
```

- The details of network (names and shapes of all weights, network structure, etc.)

  Codes building the network if possible

  or

  Visualize the network in TensorBoard

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
    saver = tf.train.import_meta_graph("model_infer.ckpt-98981.meta")
    saver.restore(sess, "model_infer.ckpt-98981")
    summary_writer = tf.summary.FileWriter('./log/', sess.graph)
```

events.out.tfevents.1559529132.dgxstation49.nvidia.com
Step 1  Load all learned weights from the saved model
reader = tf.train.NewCheckpointReader("./model.ckpt-98981")

Step 2  Create the network from scratch with TRT layer APIs, and build the engine
with trt.Builder(G_LOGGER) as builder, builder.create_network() as network:
    data = network.add_input("data", trt.float32, (3, input_h, input_w)) # add the input layer
    # add the convolution layer
    w = reader.get_tensor("resnet_v1_50/conv1/weights")
    conv = network.add_convolution(data, out_channel, (kernel_h,kernel_w), trt.Weights(w), trt.Weights(b))
    conv.stride = (stride, stride); conv.padding = (padding, padding)
...
    network.mark_output(outputs.get_output(0)) # mark outputs
engine = builder.build_cuda_engine(network) # build the engine

Step 3  Create the context and execute inference
# The TF’s input [NHWC] should be transposed to TRT format [NCHW]
with engine.create_execution_context() as context:
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]  
    context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]

https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/python_api/index.html
Tips 1: Tensor format

TensorFlow [NHWC] → TensorRT [NCHW]

The TF’s input should be transposed to TRT’s explicitly, so is the output.

```python
im = cv2.imread("test.jpg")[:, :, :, -1]
img, (ratio_h, ratio_w) = resize_image(im)
img = img.astype(np.float32)
img = mean_image_subtraction(img)
img = np.transpose(img, (2, 0, 1))
img = np.array([img])

with engine.create_execution_context() as context:
    # execute inference
    img = img.ravel()
    np.copyto(inputs[0].host, img)
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]
context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]
```
Tips 2: Weight format

**CONV**: TensorFlow [RSCK] → TensorRT [KCRS]

**RSCK**: [filter_height, filter_width, in_channel, out_channel]

**KCRS**: [out_channel, in_channel, filter_height, filter_width]

```python
w = reader.get_tensor("resnet_v1_50/conv1/weights")
w = w.transpose(3,2,0,1).reshape(-1) #RSCK->KCRS
b = np.zeros(out_channel, dtype=np.float32)
conv = network.add_convolution(inputs, out_channel, (kernel_size,kernel_size),
                               trt.Weights(w), trt.Weights(b))
conv.stride = (stride, stride)
conv.padding = (padding, padding)
```

**FC**: TensorFlow [CK] → TensorRT [KC]
Tips 3: SAME padding

SAME padding in TF may lead to asymmetric padding.

```python
net = slim.max_pool2d(net, [3, 3], stride=2, padding='SAME', scope='pool1')
```

Input map (one channel): 336x640   →   Output map (one channel): 168x320

\[
h_{output} = \left( h_{input} - h_{kernel} + h_{pad} \right) \div h_{stride} + 1 \rightarrow 168 = \left( 336 - 3 + 1 \right) \div 2 + 1
\]
Tips 4: Batch normalization

$$bn(x) = \frac{x - \text{mean}}{\sqrt{\text{var} + \varepsilon}} \cdot \text{gamma} + \text{beta} \quad \longrightarrow \quad \text{output} = (\text{input} \cdot \text{scale} + \text{shift})^{\text{power}}$$

\[
\begin{align*}
\text{scale} &= \frac{\text{gamma}}{\sqrt{\text{var} + \varepsilon}} \\
\text{shift} &= -\frac{\text{mean}}{\sqrt{\text{var} + \varepsilon}} \cdot \text{gamma} + \text{beta} \quad \text{power} = 1
\end{align*}
\]

# load gamma, beta, moving_mean and moving variance with CKPT reader
gamma = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/gamma")
beta = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/beta")
mean = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/moving_mean")
var = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/moving_variance")

# calculate the parameters and apply the scale layer
scale = gamma / np.sqrt(var + 1e-5)
shift = -mean / np.sqrt(var + 1e-5) * gamma + beta
power = np.ones(out_channel, dtype=np.float32)
bn = network.add_scale(conv.get_output(0), trt.ScaleMode.CHANNEL, trt.Weights(shift),
trt.Weights(scale), trt.Weights(power))
TRT UFF PARSER

Step 1  Convert the pb model to the uff model
convert-to-uff model.pb

Step 2  Parse the uff model and create the engine
with trt.Builder(G_LOGGER) as builder, builder.create_network() as network, trt.UffParser() as parser:
    builder.max_batch_size = 1
    builder.max_workspace_size = 1<<30
    parser.register_input("input_images", (3, 672, 1280))
    parser.register_output("feature_fusion/Conv_7/Sigmoid")
    parser.register_output("feature_fusion/concat_3")
    parser.parse("./model.uff", network)
    engine = builder.build_cuda_engine(network)

Step 3  Create context and execute inference
# The TF’s input [NHWC] should be transposed to TRT format [NCHW], no need for output
with engine.create_execution_context() as context:
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]
    context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]
### PERFORMANCE

**V100, FP32, ICDAR2015 TestSet 672x1280**

<table>
<thead>
<tr>
<th>Batchsize</th>
<th>TF Slim</th>
<th>TFTRT</th>
<th>TRT API</th>
<th>TRT Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.4</td>
<td>62.15</td>
<td>75.02</td>
<td>75.23</td>
</tr>
<tr>
<td>4</td>
<td>57.78</td>
<td>73.88</td>
<td>85.13</td>
<td>85.45</td>
</tr>
<tr>
<td>16</td>
<td>63.57</td>
<td>77.18</td>
<td>88.47</td>
<td>88.18</td>
</tr>
</tbody>
</table>

- Increasing batchsize (up to 16) improves FPS on single V100.
- TRT API and TRT Parser are more efficient in FPS than TFTRT here.
- The performances of TRT API and TRT Parser are almost the same.
CONCLUSION

- TFTRT is easy and convenient to use for TF model, but with limited acceleration now.

- TRT API and TRT UFF Parser are able to achieve better performance than TFTRT.

- TRT UFF Parser is constrained by supported ops in TRT unless adding plugins.

- TRT API is more flexible to create the network, but may lead to more work.
RESOURCES

• EAST: An Efficient and Accurate Scene Text Detector
  https://arxiv.org/abs/1704.03155

• EAST implement in TF
  https://github.com/argman/EAST

• TFTRT user guide

• TRT developer guide

• TRT API guide
  https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/index.html
Acknowledgement:
Chandler Zhou, Gary Ji, Xipeng Li @ Devtech, and Rita Zhang.