DEVELOPING REAL-TIME NEURAL NETWORKS FOR JETSON

John Welsh, 3/31/2020
OVERVIEW

What is Jetson?
Hardware, software, and ecosystem. Enabling AI at the edge.

Optimizing with TensorRT
From PyTorch to TensorFlow. Workflows for optimizing existing models. The benefits on Jetson.

Designing for real-time
Pragmatic tips to consider when creating neural networks for a real-time task.
WHAT IS NVIDIA JETSON?
JETSON AI COMPUTER LINEUP

AI Platform for Entry, Mainstream, and Fully Autonomous Edge Devices

- **JETSON NANO**
  - 0.5 TFLOPS (FP16)
  - 5-10W
  - 45mm x 70mm
  - $129

- **JETSON TX2 series (TX2, TX2 4GB, TX2i*)**
  - 1.3 TFLOPS (FP16)
  - 7.5-15W*
  - 50mm x 87mm
  - Starting at $249

- **JETSON XAVIER NX**
  - 6 TFLOPS (FP16) | 21 TOPS (INT8)
  - 10-15W
  - 45mm x 70mm
  - $399

- **JETSON AGX XAVIER series (AGX Xavier, Xavier ind.)**
  - 20-32 TOPS (INT8)
  - 5.5-11 TFLOPS (FP16)
  - 10-30W*
  - 100mm x 87mm
  - Starting at $899

One Software Architecture
JETSON ECOSYSTEM

Strong and growing

- Open source projects
  - Jetson Projects
  - NVIDIA-AI-IOT GitHub (github.com/NVIDIA-AI-IOT)
- Developer forums
- Developer kits and third party carrier boards
- Camera ecosystem partners
JETRACER: EXAMPLE APPLICATION / WORKFLOW

github.com/NVIDIA-AI-IOT/jetracer
PYTORCH TO TENSORRT
TENSORRT

GoogLeNet

Vertical fusions

Horizontal Fusions

Multiple Conv2d with single outputs -> Single Conv2d multiple outputs

Platform specific optimizations

Reduced precision

Auto kernel selection
TORCHVISION PACKAGE

Many models pre-trained on ImageNet

AlexNet, ResNet, DenseNet, MobileNet V2, to name a few

Many datasets, transformations, and utilities for vision tasks

Easy to extend and modify models for new tasks

Models largely supported by torch2trt

github.com/pytorch/vision
PyTorch => ONNX => TensorRT

**BUILD TIME**

- PyTorch Module
- Serializable Weights
- ONNX
  - Serializable
- Network
  - Serializable

**RUN TIME**

- Engine
  - Serializable
  - Platform Specific
- Execution Context
  - Runtime memory allocated

PyTorch's Tracer (Python)

TensorRT's Parser (Python or C++)

Platform specific Optimization

Platform specific Optimization
Export to ONNX using PyTorch

Uses PyTorch’s tracer to export to convert program to Graph
Graph is converted to ONNX format, serialized, and saved
github.com/onnx/onnx-tensorrt

```python
import torch
from torchvision.models import googlenet

model = googlenet(…).cuda().eval()
x = torch.ones(1, 3, 224, 224).cuda()
torch.onnx.export(model, x, 'googlenet.onnx')
```
Onnx-tensorrt
deserialize, optimize, run

Parses ONNX file
Builds optimized TensorRT engine
Implements TensorRT ONNX backend using engine
Allows simple execution on numpy arrays

```python
import onnx
import onnx_tensorrt.backend as backend
import numpy as np

model = onnx.load('googlenet.onnx')

engine = backend.prepare(model, device='CUDA:1')

x = np.zeros(32, 3, 224, 224)

y = engine.run(x)[0]
```
TORCH2TRT

github.com/NVIDIA-AI-IOT/torch2trt
Data is executed through network

“Conversion Hooks” construct network using TensorRT Python API

Engine is built using optimization parameters passed to torch2trt

TRTModule is returned, which is functionally equivalent to original PyTorch Module

```python
import torchrom torch2trt import torch2trt
from torchvision.models import googlenet

model = googlenet(...).cuda().eval()
x = torch.ones(1, 3, 224, 224).cuda()
model_trt = torch2trt(model, [x])
```
torch2trt

**execution**

Nearly same as PyTorch module

Currently, dimensions must match those provided during conversion

Batch size must not exceed max_batch_size

```python
x = torch.randn(1, 3, 224, 224).cuda()

y = model(x)
y_trt = model_trt(x)

torch.max(torch.abs(y - y_trt))
```
PyTorch/torch2trt

basics timing

Can easily profile using time library

*Be careful!* PyTorch GPU calls are asynchronous.

TRTModule is bound to PyTorch stream, use PyTorch to

synchronous

```python
import time

# benchmark throughput

t0 = time.time()

torch.cuda.current_stream().synchronize()

for i in range(100):
    y = model_trt(x)

torch.cuda.current_stream().synchronize()

t0 = time.time()

print(100.0 / (t1 - t0))
```
## Jetson Throughput (FPS) by Framework

<table>
<thead>
<tr>
<th>Platform (Precision)</th>
<th>Nano (FP32)</th>
<th>AGX Xavier (FP32)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Framework</strong></td>
<td>PyTorch</td>
<td>TensorRT</td>
</tr>
<tr>
<td>googlenet</td>
<td>22.2</td>
<td>54.3</td>
</tr>
<tr>
<td>resnet18</td>
<td>30.5</td>
<td>52.5</td>
</tr>
<tr>
<td>resnet50</td>
<td>11.2</td>
<td>18.7</td>
</tr>
<tr>
<td>densenet121</td>
<td>10</td>
<td>21.9</td>
</tr>
</tbody>
</table>
torch2trt

**reduced precision**

Supports FP16 / INT8 depending on platform
Input and output binding data types remain the same (match input data type)
Internal precision of layers determined by TensorRT builder

```python
# fp16 (internally)
x = torch.zeros(1, 3, 224, 224).cuda()
model_trt = torch2trt(model, [x], fp16_mode=True)

# fp16 (bindings also)
x = torch.zeros(1, 3, 224, 224).cuda().half()
model_trt = torch2trt(model.half(), [x], fp16_mode=True)

# int8
model_trt = torch2trt(model, [x], int8_mode=True)
```
## JETSON THROUGHPUT (FPS) BY PRECISION

<table>
<thead>
<tr>
<th>Platform</th>
<th>Nano</th>
<th>AGX Xavier</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>googlenet</td>
<td>54.3</td>
<td>90.3</td>
</tr>
<tr>
<td></td>
<td>316</td>
<td>527</td>
</tr>
<tr>
<td></td>
<td>672</td>
<td></td>
</tr>
<tr>
<td>resnet18</td>
<td>52.5</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>339</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>1030</td>
<td></td>
</tr>
<tr>
<td>resnet50</td>
<td>18.7</td>
<td>37.2</td>
</tr>
<tr>
<td></td>
<td>117</td>
<td>321</td>
</tr>
<tr>
<td></td>
<td>458</td>
<td></td>
</tr>
<tr>
<td>densenet121</td>
<td>21.9</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>114</td>
<td>164</td>
</tr>
<tr>
<td></td>
<td>230</td>
<td></td>
</tr>
</tbody>
</table>
# JETSON SUPPORT MATRIX

<table>
<thead>
<tr>
<th></th>
<th>Nano</th>
<th>TX2</th>
<th>Xavier NX</th>
<th>AGX Xavier</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory</strong></td>
<td>4GB</td>
<td>8GB</td>
<td>8GB</td>
<td>8-32GB</td>
</tr>
<tr>
<td><strong>Fp16 Support</strong></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Int8 Support</strong></td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Deep Learning Accelerators</strong></td>
<td>NONE</td>
<td>NONE</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
torch2trt

Batch size

Batching reduces relative overhead, improves throughput
Specified by parameter, not input data
Runtime batch size must not exceed value

```python
x = torch.zeros(1, 3, 224, 224).cuda()
model_trt = torch2trt(
    model, [x], max_batch_size=8
)
y = torch.zeros(8, 3, 224, 224).cuda()
z = model_trt(y)
```
<table>
<thead>
<tr>
<th>Platform (Precision)</th>
<th>Nano (FP16)</th>
<th>AGX Xavier (FP16)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Batch Size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>googlenet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>90.4</td>
<td>96.9</td>
</tr>
<tr>
<td></td>
<td>11.5</td>
<td>21.2</td>
</tr>
<tr>
<td>resnet18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>90.8</td>
<td>98.4</td>
</tr>
<tr>
<td></td>
<td>11.5</td>
<td>20.8</td>
</tr>
<tr>
<td>resnet50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>34.7</td>
<td>39.4</td>
</tr>
<tr>
<td></td>
<td>29.5</td>
<td>51.6</td>
</tr>
<tr>
<td>densenet121</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>42.1</td>
<td>44.9</td>
</tr>
<tr>
<td></td>
<td>24.7</td>
<td>46</td>
</tr>
</tbody>
</table>
torch2trt
rename bindings

By default, inputs are named input_0, input_1, ... in order.
Outputs are named output_0, output_1, ...
Can re-map input / output names if needed

```python
x = torch.zeros(1, 3, 224, 224).cuda()
model_trt = torch2trt(model, [x],
    input_names=['image'],
    output_names=['logits'])
```
torch2trt

By default, calibrates in input data

Tracing ignores batch, but calibration uses all data in batch

Can override default calibration algorithm (see TensorRT Python API for options)

```python
# calibrate on random data
x = torch.randn(32, 3, 224, 224).cuda()

model_trt = torch2trt(model, [x], int8_mode=True)

# specify calibration algorithm
model_trt = torch2trt(model, [x], int8_mode=True, int8_calib_algorithm=…)
```
torch2trt

Dynamically loads input data to support larger datasets
Calibration dataset provides only inputs, excluding batch

int8 calibration (more data)

```python
# define input dataset class
class ImageFolderDataset:
    def __init__(self, folder):
        self.paths = glob.glob(...)

    def __len__(self):
        return len(self.paths)

    def __getitem__(self, idx):
        path = self.paths[idx]
        # load image to CxHxW tensor
        return [image]

# calibrate on image folder
calib_dataset = ImageFolderDataset('images')
x = torch.zeros(1, 3, 224, 224).cuda()
model_trt = torch2trt(model, [x], int8_mode=True,
int8_calib_dataset=calib_dataset)
```
**torch2trt**

**int8 calibration (multiple inputs)**

Some modules require multiple inputs
Specified in order they are fed to module
(although not GoogLeNet)

class StereoImageDataset():
    ...
    def __getitem__(self, idx):
        ...
        return [left_image, right_image]

# calibrate multiple input model
calib_dataset = StereoDataset(...)

left = torch.zeros(1, 3, 224, 224).cuda()
right = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(model,[left, right],
                        int8_mode=True,
                        int8_calib_dataset=calib_dataset)
torch2trt

save / load

Same as PyTorch module
Allows TRTModule to replace PyTorch submodule
Network is dropped when saved (since it is not serializable)

# save state dict
torch.save(model_trt.state_dict(), 'model_trt.pth')

# load state dict
From torch2trt import TRTModule()

model_trt = TRTModule()
model_trt.load_State_dict(
    torch.load('model_trt.pth'))
EXECUTION AND STORAGE PARITY

Allows partial conversion of modules

# replace and save
parent.supported = torch2trt(parent.supported, ...)
torch.save(parent.state_dict(), ...)

# replace and load
parent.supported = TRTModule()
torch.load_state_dict(…)

```python
# replace and save
parent.supported = torch2trt(parent.supported, ...)
torch.save(parent.state_dict(), ...)

# replace and load
parent.supported = TRTModule()
torch.load_state_dict(…)
```
SAVING FOR C++
Same as TensorRT Python API!

with open('model.engine', 'wb') as f:
    f.write(model_trt.engine.serialize())
torch2trt

custom converter

torch2trt is easy to modify
Define converter with @tensorrt_converter
Converter takes a “ConversionContext”
  ctx.network - TensorRT network being constructed
  ctx.method_args - Arguments to PyTorch method
  ctx.method_kwarg - Keyword args to PyTorch method
  ctx.method_return - Return value of PyTorch method
Converter uses TensorRT Python API to extend network
Converter must set _trt attribute of relevant torch outputs

@tensorrt_converter('torch.relu')
def convert_relu(ctx):
    input = ctx.method_args[0]
    output = ctx.method_return
    trt_layer = ctx.network.add_activation(
        input=input._trt,
        type=trt.ActivationType.RELU
    )
    output._trt = trt_layer.get_output(0)
NETWORK VISUALIZATION
What layers were added to the network?

Convert network to GraphViz “Dot” format

Useful for debugging

Easily render as PDF

```python
from torch2trt.utils import trt_network_to_dot_graph
dot = trt_network_to_dot_graph(model_trt.network)
dot.render('googlenet.gv', view=True)
```
NETWORK VISUALIZATION

How are layers mapped?

Use TensorRT profiler! (next slide)
Each line in output is single layer
Horizontal 1x1 convolutions fused
Batch Norms (Scale) fused
Activations fused
torch2trt
TensorRT profiling

Adds fine-grained profiling of internal TensorRT layers
Prints to stdout
Model execution becomes synchronous

```
x = torch.zeros(1, 3, 224, 224).cuda()
model_trt.enable_profiling()
y = model_trt(x)
```
PyTorch/torch2trt
CUDA profiling

Capture all CUDA runtime calls in region
Dump files for NVIDIA Visual Profiler

```python
torch.cuda.profiler.init('googlenet.nvvp', output_mode='csv')

# collect region using context manager
with torch.cuda.profiler.profile():
    y = model_trt(x)
    torch.cuda.current_stream().synchronize()

# collect region using start/stop
torch.cuda.current_stream().synchronize()
torch.cuda.profiler.start()
    y = model_trt(x)
    torch.cuda.current_stream().synchronize()
torch.cuda.profiler.stop()
```
PYTORCH VISUAL PROFILE

Low GPU Utilization

Many kernel Invocations
TENSORRT VISUAL PROFILE

High GPU Utilization

Few Kernel Invocations
TENSORFLOW TO TENSORRT
OPTIMIZING TENSORFLOW

What are our options?

- TF-TRT (TensorRT integration in TensorFlow)
  - Runs like a normal TensorFlow graph
  - Unsupported operations fall-back to TensorFlow

- TensorFlow -> UFF -> TensorRT
  - Convert TensorFlow graph to UFF format
  - Parse UFF file and optimize with TensorRT
  - Requires TensorRT Plugins for unsupported parts
SINGLE SHOT DETECTOR

Case study (TF-TRT)

- Sourced from TensorFlow object detection API
- CNN Backbone
  - Supported, except ReLU 6 (at the time)
- Anchor box parsing
  - Fall back to TF
- Non-maximum suppression
  - Fall back to TF
  - Native TF was slow... repetitive unnecessary CPU/GPU copies

 relu6(x) = relu(x) - relu(x - 6)  
 fall back  
 fall back, force CPU
**TensorFlow/TF-TRT**

**TensorFlow profiler**

- Execute TensorFlow graph enabling tracing
- Export metadata in chrome trace format
- Visualize with chrome browser
  - Easily spot data copies, layer calls, layer devices
- We used this to find a CPU->GPU copy bottleneck

```python
options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
run_metadata = tf.RunMetadata()
Sess.run(..., options=options, run_metadata=run_metadata)
run_timeline = timeline.Timeline(run_metadata.step_stats)
Chrome_trace = run_timeline.generate_chrome_trace_format()
```
TENSORFLOW PROFILE TRACE
TF-TRT

Optimize frozen graph

One call: "create_inference_graph"
Input is frozen graph, with all TensorFlow layers
Output is frozen graph, with sub-graphs as TensorRT blocks
Minimum segment size is used to control granularity
prevent "small" engines with non-negligible overhead

```python
frozen_graph = tf.GraphDef()
with open('frozen_inference_graph.pb', 'rb') as f:
    frozen_graph.ParseFromString(f.read())

trt_graph = trt.create_inference_graph(
    input_graph_def=frozen_graph,
    outputs=['detection_boxes',
             'detection_classes',
             'detection_scores',
             'num_detections'],
    max_batch_size=1,
    max_workspace_size=1 << 25,
    precision_mode='FP16',
    minimum_segment_size=50)
```
TF-TRT

Execute graph

Set `allow_growth` to prevent TensorFlow from hogging Jetson memory.

```python
# configure session to allow growth for memory
tf_config = tf.ConfigProto()
tf_config.gpu_options.allow_growth = True
tf_sess = tf.Session(config=tf_config)

# load optimized graph
tf.import_graph_def(trt_graph, name='')

# execute graph as normal tensorflow ...
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Input Size</th>
<th>TF-TRT TX2</th>
<th>TF TX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssd_mobilenet_v1_coco</td>
<td>300x300</td>
<td>50.5ms</td>
<td>72.9ms</td>
</tr>
<tr>
<td>ssd_inception_v2_coco</td>
<td>300x300</td>
<td>54.4ms</td>
<td>132ms</td>
</tr>
</tbody>
</table>
DESIGNING FOR REAL-TIME
PRAGMATIC CONSTRAINTS

For real-time deployment on Jetson

- Avoid data dependent CNN execution like two-stage detectors (when appropriate)
  - Typically, this will keep runtime and memory nearly static

- Use TensorRT supported layers when possible
  - Using just one framework can reduce memory consumption
  - More possible fusions, fewer unnecessary type casting / reformatting

- Lightweight post-processing / parsing
  - Similar to (1), to ensure near-constant runtime
POSE DETECTION

Case study

- Top performing methods commonly include
  - Two stage detectors
  - Ensemble networks
- These methods are usually computationally expensive
  - Two Stage scale’s with number of objects in image
CENTERNET
Near static runtime

- Single CNN produces feature maps
- Objects parse by finding peak of heatmap
- Other semantics then parsed
- No second large CNN execution

PART AFFINITY FIELDS
Near static runtime

- Single CNN produces two feature maps
  - Confidence Map
  - Part affinity field
- Part x,y coordinates proposed from local maxima of confidence maps
- Part associate scores produced by integrating between parts
- Assignment algorithm applied to associate parts
  - <1ms on CPU typically

TRT-POSE: REAL-TIME POSE DETECTION

github.com/NVIDIA-AI-IOT/trt_pose

- Resnet18_baseline_att
  - Resnet18 well optimized by TensorRT for Jetson
  - 3x Deconvolution at 4x4 pixel natively supported by TRT
- CMAP / PAF post processing
  - Low post-processing runtime
- ~22 FPS Jetson Nano
USEFUL EXTRAS

- Torchvision package
  - Many TensorRT ready pre-trained backbone architectures
  - Easy to use / extend
  - github.com/pytorch/vision

- Segmentation_models.pytorch
  - Many TensorRT ready multi-scale pre-trained backbone architectures
  - Easy to use / extend
  - github.com/qubvel/segmentation_models.pytorch

- Jetson Benchmarks
  - Various reproducible benchmarks for tasks like Object Detection with TensorRT. Including DLA.
  - github.com/NVIDIA-AI-IOT/jetson_benchmarks