S8495: DEPLOYING DEEP NEURAL NETWORKS AS-A-SERVICE USING TENSORRT AND NVIDIA-DOCKER

Prethvi Kashinkunti, Solutions Architect
Alec Gunny, Solutions Architect
Deploying Deep Learning Models
- Current Approaches
- Production Deployment Challenges

NVIDIA TensorRT as a Deployment Solution
- Performance, Optimizations and Features

Deploying DL models with TensorRT
- Import, Optimize and Deploy
  - TensorFlow image classification
  - PyTorch LSTM
  - Caffe object detection

Inference Server Demos

Q&A
WHAT DO I DO WITH MY TRAINED DL MODELS?

Gain insight from data

• Congrats, you’ve just finished trained your DL model (and it works)!

• My DL serving solution wish list:
  • Can deliver sufficient performance ← key metric!
  • Is easy to set up
  • Can handle models for multiple use cases from various training frameworks
  • Can be accessed easily by end-users
CURRENT DEPLOYMENT WORKFLOW

TRAINING

- Training Data
- Data Management
- Training
- Model Assessment
- Trained Neural Network

DEPLOYMENT

1. Deploy framework or custom CPU-Only application
2. Deploy training framework on GPU
3. Deploy custom application using NVIDIA DL SDK

CUDA, NVIDIA Deep Learning SDK (cuDNN, cuBLAS, NCCL)
DEEP LEARNING AS - A - (EASY) SERVICE

Proof of Concept

• Opportunities for optimizing our deployment performance
  1. High performance serving infrastructure
  2. Improving model inference performance ← we’ll start here

• DL-aas Proof-of-Concept:
  • Use NVIDIA TensorRT to create optimized inference engines for our models
    • Freely available as a container in the NVIDIA GPU Cloud (ngc.nvidia.com)
    • More details to come on TensorRT...
  • Create a simple Python Flask application to expose models via REST endpoints
DEEP LEARNING AS - A - (EASY) SERVICE

Architecture Diagram

NVIDIA GPU Cloud container:
(nvcr.io/nvidia/tensorrt:18.01-py2)

Server with GPU

/classify (Keras/TF)  /generate (PyTorch)  /detect (Caffe)

(TensorRT Inference Engines)

(RESTful API endpoints from Python Flask app)

End Users: Send inference request, receive response from server
NVIDIA TENSORRT OVERVIEW
NVIDIA TENSORRT
Programmable Inference Accelerator

FRAMEWORKS
- Caffe2
- PaddlePaddle
- Chainer
- PyTorch
- TensorFlow
- mxnet
- Theano

GPU PLATFORMS
- TESLA P4
- JETSON TX2
- DRIVE PX 2
- TESLA V100
- NVIDIA DLA

TensorRT
Optimizer
Runtime

developer.nvidia.com/tensorrt
TENSORRT OPTIMIZATIONS

➢ Optimizations are completely automatic
➢ Performed with a single function call

```python
engine = trt.utils.uff_to_trt_engine(G_LOGGER,
    uff_model,
    parser,
    INFEERENCE_BATCH_SIZE,
    1<<20,
    trt.infer.DataType.FLOAT)
```
TENSORRT DEPLOYMENT WORKFLOW

**Step 1: Optimize trained model**

1. Trained Neural Network → Import Model → TensorRT Optimizer → Serialize Engine → Optimized Plans

**Step 2: Deploy optimized plans with runtime**

1. Optimized Plans → De-serialize Engine → Deploy Runtime → Data center, Automotive, Embedded

---

---
IMPORTING MODELS TO TENSORRT
TENSORRT DEPLOYMENT WORKFLOW

Step 1: Optimize trained model

- Trained Neural Network
  - Import Model
  - TensorRT Optimizer
  - Serialize Engine
  - Optimized Plans

Step 2: Deploy optimized plans with runtime

- Optimized Plans
  - De-serialize Engine
  - Deploy Runtime
  - TensorRT Runtime Engine
  - Data center
  - Automotive
  - Embedded
MODEL IMPORTING PATHS

➢ AI Researchers
➢ Data Scientists

Caffe ➢ TensorFlow ➢ Other Frameworks
Python/C++ API
Model Importer

Network Definition API

Runtime inference
C++ or Python API

developer.nvidia.com/tensorrt
VGG19: KERAS/TF
Image classification

- Model is Keras VGG19 model pretrained on ImageNet, finetuned for flowers dataset from TF Slim
- Using TF backend, freeze graph to convert weight variables to constants
- Import into TensorRT using built-in TF->UFF->TRT parser

```python
def generate_engine(filename, use_fp16):
    # Load your newly created Tensorflow frozen model and convert it to UFF
    uff_model = uff.from_tensorflow_frozen_model("keras_vgg19_frozen_model.pb", OUTPUT_LAYERS)

    # Create a UFF parser to parse the UFF file created from your TF Frozen model
    parser = uffparser.create_uff_parser()
    parser.register_input(INPUT_LAYERS[0], (INPUT_C,INPUT_H,INPUT_W), 0)
    parser.register_output(OUTPUT_LAYERS[0])

    # Build your TensorRT inference engine
    if(use_fp16):
        engine = trt.utils.uff_to_trt_engine(G_LOGGER, uff_model, parser, INFERENCCE_BATCH_SIZE, 1<<20, trt.infer.DataType.HALF)
        precision = "fp16"
    else:
        engine = trt.utils.uff_to_trt_engine(G_LOGGER, uff_model, parser, INFERENCCE_BATCH_SIZE, 1<<20, trt.infer.DataType.FLOAT)
        precision = "fp32"

    # Serialize TensorRT engine to a file for when you are ready to deploy your model.
    save_path = "/keras_vgg19_b"+str(INFERENCCE_BATCH_SIZE) + "+"+ precision + ".engine"

    trt.utils.write_engine_to_file(save_path, engine.serialize())
```
CHAR_RNN: PYTORCH

Text Generation

- Model is character-level RNN model (using LSTM cell) trained with PyTorch
  - Training data: .py files from PyTorch source code
- Export PyTorch model weights to Numpy, permute to match FICO weight ordering used by cuDNN/TensorRT
- Import into TensorRT using Network Definition API

```python
shuffle_weight = lambda w: np.concatenate([w[hidden_size:2*hidden_size], w[:hidden_size], w[2*hidden_size:]], axis=0)

rnn_weights, rnn_biases = [], []
for layer in range(decoder.n_layers):
    w_l = shuffle_weight(weights['rnn.weight_ih_l{}'.format(layer)].cpu().numpy().reshape(-1))
    wb_l = shuffle_weight(weights['rnn.bias_ih_l{}'.format(layer)].cpu().numpy())
    u_l = shuffle_weight(weights['rnn.weight_hh_l{}'.format(layer)].cpu().numpy().reshape(-1))
    ub_l = shuffle_weight(weights['rnn.bias_hh_l{}'.format(layer)].cpu().numpy())
    rnn_weights.append(np.concatenate([w_l, u_l]))
    rnn_biases.append(np.concatenate([wb_l, ub_l]))

rnn_weights, rnn_biases = np.concatenate(rnn_weights), np.concatenate(rnn_biases)
if use_fp16:
    rnn_weights = rnn_weights.astype(np.float16)
    rnn_biases = rnn_biases.astype(np.float16)

rnn = network.add_rnn(data, decoder.n_layers,
                       hidden_size,
                       1,
                       trt.infer.RNNOperation_KLSTM,
                       trt.infer.RNNInputMode_KLINEAR,
                       trt.infer.RNNDirection_KUNIDIRECTION,
                       rnn_weights,
                       rnn_biases)
```
SINGLE SHOT DETECTOR: CAFFE

Object Detection

- Model is SSD object detection model trained with Caffe
  - Training data: Annotated traffic intersection data
- Network includes several layers unsupported by TensorRT:
  - Permute, PriorBox, etc
  - Requires use of custom layer API!
- Use built-in Caffe network parser to import network along with custom layers

```c++
// create the builder
IBuilder* builder = createInferBuilder(gLogger);

// parse the caffe model to populate the network, then set the outputs
INetworkDefinition* network = builder->createNetwork();
ICaffeParser* parser = createCaffeParser();
parser->setPluginFactory(pluginFactory);

std::cout << "Begin parsing model...
const IBlobNameToTensor* blobNameToTensor = parser->parse(locateFile(
deployFile).c_str(),
    locateFile(modelFile).c_str(),
    *network,
    halfmode ? DataType::kHALF:DataType::kFLOAT);
std::cout << "End parsing model...
// specify which tensors are outputs
for (auto& s : outputs)
    network->markOutput(*blobNameToTensor->find(s.c_str()));

// Build the engine
builder->setMaxBatchSize(maxBatchSize);
builder->setMaxWorkspaceSize(14 << 24);
builder->setHalf2Mode(halfmode);

std::cout << "Begin building engine...
ICudaEngine* engine = builder->buildCudaEngine(*network);
assert(engine);
```
DESIGNING THE INFERENCE SERVER

Putting it all together...

- Using TensorRT Python API, we can wrap all of these inference engines together into a simple Flask application
  - Similar example code provided in TensorRT container
- Create three endpoints to expose models:
  - /classify
  - /generate
  - /detect

```python
# Network 1: Keras image classification model
PLAN_classification = "../build_engines/keras_vgg19_b1_fp16.engine"
LABELS = ["daisy", "dandelion", "roses", "sunflowers", "tulips"]

# Network 2: PyTorch LSTM based charRNN model
PLAN_charRNN = "../build_engines/pytorch_source_512.pt.engine"
embedding_mtrx = np.load("../build_engines/pytorch_source_512.pt_embedding.npy")
hidden_size = 512
n_layers = 2
output_size = 100
all_characters = string.printable

# Network 3: Caffe SSD model
PLAN_detection = "/workspace/tensorrt/bin/ssd_b1_fp32.engine"
factory = plugins.SSDPluginFactory()
INPUT_H = 300
INPUT_W = 300

# Reload serialized TRT engines onto the GPU
engine_classification = Engine(PLAN=PLAN_classification, postprocessors={
    "dense 2/Softmax": get_classification})
print("loaded classification engine")

engine_charRNN = trt.utils.load_engine(G_LOGGER, PLAN_charRNN)
context_charRNN = engine_charRNN.create_execution_context()
print("loaded charRNN engine")

engine_detection = trt.utils.load_engine(G_LOGGER, PLAN_detection, factory)
context_detection = engine_detection.create_execution_context()
print("loaded detection engine")

# Define Inference Web service
app = Flask(__name__)
```
SCALING IT UP
DESIGNING THE INFERENCE SERVER

Easy improvements for better perf

- Our DL-aas proof-of-concept works, yay!
- One main drawback: single threaded serving
- Instead, can use tools like Gunicorn & Nginx to easily scale your inference workload across more compute
  - Multithreaded containerized workers tied to their own GPU
  - Straightforward to integrate w/ Flask app
GETTING CLOSER TO PRODUCTION
Areas for potential improvement

• Previous example mostly addresses our needs, but has room for improvement...

• Potential improvements:
  
  • Batching of requests
  
  • Autoscaling of compute resources based on workload
  
  • Improving performance of pre/post processing around TensorRT inference
    
    • E.g. image resizing
  
  • Better UI/UX for client side
TENSORRT KEY TAKEAWAYS

✓ Generate optimized, deployment-ready runtime engines for low latency inference

✓ Import models trained using Caffe or TensorFlow or use Network Definition API

✓ Deploy in FP32 or reduced precision INT8, FP16 for higher throughput

✓ Optimize frequently used layers and integrate user defined custom layers
LEARN MORE

Helpful Links

- GPU Inference Whitepaper:

- Blogpost on using TensorRT 3.0 for TF model inference:
  - https://devblogs.nvidia.com/tensorrt-3-faster-tensorflow-inference/

- TensorRT documentation:
  - http://docs.nvidia.com/deeplearning/sdk/index.html#inference
Q&A