pytorch_onnx_trt

July 11, 2019

1 Define a model that serves as an example

[64]:
```
import torch

class CNN(torch.nn.Module):
    def __init__(self, num_classes=10,):
        super(CNN, self).__init__()
        self.layer1 = torch.nn.Conv2d(3,16,3)
        self.layer2 = torch.nn.Conv2d(16,64,5)
        self.relu = torch.nn.ReLU()

        # TAKE CARE HERE
        # Ceil_mode must be False, because onnx eporter does NOT support␣
↪ceil_mode=True
        self.max_pool = torch.nn.MaxPool2d(kernel_size=3, stride=1,␣
↪ceil_mode=False)
        self.avg_pool = torch.nn.AdaptiveAvgPool2d((1,1))

        self.fc = torch.nn.Linear(64,num_classes)
        self.batch_size_onnx = 0
        # FLAG for output ONNX model
        self.export_to_onnx_mode = False
    def forward_onnx(self, X_in):
        print("Function forward_onnx called! \n")
        x = self.layer1(X_in)
        x = self.relu(x)
        x = self.max_pool(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.avg_pool(x)
        assert self.batch_size_onnx > 0
        length_of_fc_layer = 64 # For exporting an onnx model that fit the␣
↪TensorRT, processes here should be DETERMINISITC!
        x = x.view(self.batch_size_onnx, length_of_fc_layer) #
        x = self.fc(x)
        return x
```

```
    def forward_default(self, X_in):
        print("Function forward_default called! \n")
        x = self.layer1(X_in)
        x = self.relu(x)
        x = self.max_pool(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.avg_pool(x)

        # Such an operationt is not deterministic since it would depend on the␣
↪input and therefore would result in errors
        length_of_fc_layer = x.size(1)
        x = x.view(-1, length_of_fc_layer)

        x = self.fc(x)
        return x

    def __call__(self, *args,**kargs):
        if self.export_to_onnx_mode:
            return self.forward_onnx(*args,**kargs)
        else:
            return self.forward_default(*args,**kargs)
```

2 Export a PyTorch model to an ONNX model

TensorRT does not support to PyTorch model directly, but fortunately we could take advantage of ONNX

[65]:
```
onnx_model_path = './model.onnx'
pth_model_path  = './model.pth'

# Load the model
model = CNN(10)
# Save the state dict, which will be loaded later
print('PyTorch model saved to {}\n'.format(pth_model_path))
torch.save(model.state_dict(), pth_model_path)

# This is for ONNX exporter to track all the operations inside the model
batch_size_of_dummy_input = 2 # Any size you want
dummy_input = torch.zeros([batch_size_of_dummy_input, 3, 64, 64], dtype=torch.
↪float32)

model.batch_size_onnx = batch_size_of_dummy_input
model.export_to_onnx_mode = True
input_names = [ "input" ]
```

```
output_names = [ "output"] # Multiple inputs and outputs are supported
with torch.no_grad():
    # If verbose is set to False. The information below won't displayed
    torch.onnx.export(model, dummy_input, onnx_model_path, verbose=True,␣
 ↪input_names=input_names, output_names=output_names)
print('ONNX model exported to {}\n'.format(onnx_model_path))
```

```
PyTorch model saved to ./model.pth

Function forward_onnx called!

graph(%input : Float(2, 3, 64, 64),
      %layer1.weight : Float(16, 3, 3, 3),
      %layer1.bias : Float(16),
      %layer2.weight : Float(64, 16, 5, 5),
      %layer2.bias : Float(64),
      %fc.weight : Float(10, 64),
      %fc.bias : Float(10)):
  %7 : Float(2, 16, 62, 62) = onnx::Conv[dilations=[1, 1], group=1,
kernel_shape=[3, 3], pads=[0, 0, 0, 0], strides=[1, 1]](%input, %layer1.weight,
%layer1.bias), scope: Conv2d
  %8 : Float(2, 16, 62, 62) = onnx::Relu(%7), scope: ReLU
  %9 : Float(2, 16, 60, 60) = onnx::MaxPool[kernel_shape=[3, 3], pads=[0, 0, 0,
0], strides=[1, 1]](%8), scope: MaxPool2d
  %10 : Float(2, 64, 56, 56) = onnx::Conv[dilations=[1, 1], group=1,
kernel_shape=[5, 5], pads=[0, 0, 0, 0], strides=[1, 1]](%9, %layer2.weight,
%layer2.bias), scope: Conv2d
  %11 : Float(2, 64, 56, 56) = onnx::Relu(%10), scope: ReLU
  %12 : Float(2, 64, 1, 1) = onnx::GlobalAveragePool(%11), scope:
AdaptiveAvgPool2d
  %13 : Float(2, 64) = onnx::Flatten[axis=1](%12)
  %output : Float(2, 10) = onnx::Gemm[alpha=1, beta=1, transB=1](%13,
%fc.weight, %fc.bias), scope: Linear
  return (%output)

ONNX model exported to ./model.onnx
```

3 Prepare some useful functions

[66]:
```
import pycuda.autoinit
import numpy as np
import pycuda.driver as cuda
import tensorrt as trt
import torch
```

```
TRT_LOGGER = trt.Logger() # This logger is required to build an engine


class HostDeviceMem(object):
    def __init__(self, host_mem, device_mem):
        """Within this context, host_mom means the cpu memory and device means␣
 ↪the GPU memory
        """
        self.host = host_mem
        self.device = device_mem
    def __str__(self):
        return "Host:\n" + str(self.host) + "\nDevice:\n" + str(self.device)

    def __repr__(self):
        return self.__str__()

def allocate_buffers(engine):
    inputs = []
    outputs = []
    bindings = []
    stream = cuda.Stream()
    for binding in engine:
        size = trt.volume(engine.get_binding_shape(binding)) * engine.
 ↪max_batch_size
        dtype = trt.nptype(engine.get_binding_dtype(binding))
        # Allocate host and device buffers
        host_mem = cuda.pagelocked_empty(size, dtype)
        device_mem = cuda.mem_alloc(host_mem.nbytes)
        # Append the device buffer to device bindings.
        bindings.append(int(device_mem))
        # Append to the appropriate list.
        if engine.binding_is_input(binding):
            inputs.append(HostDeviceMem(host_mem, device_mem))
        else:
            outputs.append(HostDeviceMem(host_mem, device_mem))
    return inputs, outputs, bindings, stream

def get_engine(max_batch_size=1, onnx_file_path="", engine_file_path="",\
               fp16_mode=False, int8_mode=False, save_engine=False,
              ):
    """Attempts to load a serialized engine if available, otherwise builds a new␣
 ↪TensorRT engine and saves it."""
    def build_engine(max_batch_size, save_engine):
        """Takes an ONNX file and creates a TensorRT engine to run inference␣
 ↪with"""
        with trt.Builder(TRT_LOGGER) as builder, \
                builder.create_network() as network,\
```

```
                trt.OnnxParser(network, TRT_LOGGER) as parser:

            builder.max_workspace_size = 1 << 30 # Your workspace size
            builder.max_batch_size = max_batch_size
            #pdb.set_trace()
            builder.fp16_mode = fp16_mode  # Default: False
            builder.int8_mode = int8_mode  # Default: False
            if int8_mode:
                # To be updated
                raise NotImplementedError

            # Parse model file
            if not os.path.exists(onnx_file_path):
                quit('ONNX file {} not found'.format(onnx_file_path))

            print('Loading ONNX file from path {}...'.format(onnx_file_path))
            with open(onnx_file_path, 'rb') as model:
                print('Beginning ONNX file parsing')
                parser.parse(model.read())
            print('Completed parsing of ONNX file')
            print('Building an engine from file {}; this may take a while...'.
 ↪format(onnx_file_path))
            #pdb.set_trace()
            #network.mark_output(network.get_layer(network.num_layers-1).
 ↪get_output(0)) # Riz
            #network.mark_output(network.get_layer(network.num_layers-1).
 ↪get_output(1)) # Riz

            engine = builder.build_cuda_engine(network)
            print("Completed creating Engine")

            if save_engine:
                with open(engine_file_path, "wb") as f:
                    f.write(engine.serialize())
            return engine

    if os.path.exists(engine_file_path):
        # If a serialized engine exists, load it instead of building a new one.
        print("Reading engine from file {}".format(engine_file_path))
        with open(engine_file_path, "rb") as f, trt.Runtime(TRT_LOGGER) as␣
 ↪runtime:
            return runtime.deserialize_cuda_engine(f.read())
    else:
        return build_engine(max_batch_size, save_engine)

def do_inference(context, bindings, inputs, outputs, stream, batch_size=1):
    # Transfer data from CPU to the GPU.
```

```
    [cuda.memcpy_htod_async(inp.device, inp.host, stream) for inp in inputs]
    # Run inference.
    context.execute_async(batch_size=batch_size, bindings=bindings,␣
 ↪stream_handle=stream.handle)
    # Transfer predictions back from the GPU.
    [cuda.memcpy_dtoh_async(out.host, out.device, stream) for out in outputs]
    # Synchronize the stream
    stream.synchronize()
    # Return only the host outputs.
    return [out.host for out in outputs]

def postprocess_the_outputs(h_outputs, shape_of_output):
    h_outputs = h_outputs.reshape(*shape_of_output)
    return h_outputs
```

4 Load an ONNX model and build an TensorRT engine

[67]:
```
import os
import time

onnx_model_path = './model.onnx'
pytorch_model_path = './model.pth'

# These two modes are dependent on hardwares
fp16_mode = False
int8_mode = False
trt_engine_path = './model_fp16_{}_int8_{}.trt'.format(fp16_mode, int8_mode)


max_batch_size = 1 # The batch size of input mush be smaller the max_batch_size␣
 ↪once the engine is built
x_input = np.random.rand(max_batch_size, 3, 64, 64).astype(dtype=np.float32)

# Build an engine
engine = get_engine(max_batch_size, onnx_model_path, trt_engine_path, fp16_mode,␣
 ↪int8_mode)
# Create the context for this engine
context = engine.create_execution_context()
# Allocate buffers for input and output
inputs, outputs, bindings, stream = allocate_buffers(engine) # input, output:␣
 ↪host # bindings


# Do inference
shape_of_output = (max_batch_size, 10)
```

```
# Load data to the buffer
inputs[0].host = x_input.reshape(-1)
# inputs[1].host = ... for multiple input
t1 = time.time()
trt_outputs = do_inference(context, bindings=bindings, inputs=inputs,␣
 ↪outputs=outputs, stream=stream) # numpy data
t2 = time.time()
output_from_trt_engine = postprocess_the_outputs(trt_outputs[0], shape_of_output)

# Compare with the PyTorch
pth_model = CNN(10)
pth_model.load_state_dict(torch.load(pytorch_model_path))
pth_model.cuda()

x_input_pth = torch.from_numpy(x_input).cuda()
pth_model.export_to_onnx_mode = False
t3 = time.time()
output_from_pytorch_model =  pth_model(x_input_pth)
t4 = time.time()
output_from_pytorch_model = output_from_pytorch_model.cpu().data.numpy()


mse = np.mean((output_from_trt_engine - output_from_pytorch_model)**2)
print("Inference time with the TensorRT engine: {}".format(t2-t1))
print("Inference time with the PyTorch model: {}".format(t4-t3))
print('MSE Error = {}'.format(mse))
```

```
Loading ONNX file from path ./model.onnx...
Beginning ONNX file parsing
Completed parsing of ONNX file
Building an engine from file ./model.onnx; this may take a while...
Completed creating Engine
Function forward_default called!

Inference time with the TensorRT engine: 0.0009050369262695312
Inference time with the PyTorch model: 0.0011165142059326172
MSE Error = 5.2735593007950446e-17
```

[ ]: