Transformer TensorRT Inference

- Last updated: 2021-09-06
- project path: https://github.com/MrDoghead/nmt-nemo.git

This project is to implement transformer inference using TensorRT engines. So we are not going through the transformer architecture and beam search procedure.

In this document, we will give the tensorRT installation methods and then describe how to convert a pytorch model to onnx and build tensorRt engines. Besides, we will talk the tensorRT inference for generative model. Finally, some issues and bugs are listed with their solutions.

1. TenosrRT Installation

There are 2 ways to intall tensorRT and we recomamded you to use Container Installation.

1.1 Container installation

Please check `./docker` before installation

```
bash ./docker/build.sh
```

1.2 From packages

Follow the steps below to install both GA core and Oss components

- Tensorrt GA Core

1. package https://developer.nvidia.com/nvidia-tensorrt-7x-download

   ```
    we recommanded to choose >7.0 version that match your system
   ```

2. decompress

   ```
   tar -xvzf TensorRT-7.2.3.4.Ubuntu-18.04.x86_64-gnu.cuda-11.0.cudnn8.1.tar.gz
   ```

3. add env configs

   ```
   vim ~/.bashrc
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:"/home/ubuntu/TensorRT-7.2.3.4/lib"
   source ~/.bashrc
   ```

4. install python wheels

   ```
   pip install tensorrt-7.2.3.4-cp38-none-linux_x86_64.whl
   ```

- Tensorrt Oss Components

1. download cmake

   ```
   wget https://cmake.org/files/v3.13/cmake-3.13.2.tar.gz
   tar xvf cmake-3.13.2.tar.gz
   cd cmake-3.13.2
   ```

2. install cmake

   ```
   ./bootstrap --prefix=/usr
   make
   sudo make install
   cmake --version
   ```

Note: if you meet some qt error, try to add --no-qt-gui after ./bootstrap

3. download oss

```
git clone -b release/7.2 https://github.com/nvidia/TensorRT TensorRT
cd TensorRT
git submodule update --init --recursive
```

Note: please checkout to the corresponding branch according to you GA version (here 7.2.3.4)

4. Build Oss

```
mkdir -p build && cd build
```

add `export TRT_LIBPATH='/home/ubuntu/TensorRT-7.2.3.4'` to `~/.bashrc`

```
source ~/.bashrc
cmake .. -DTRT_LIB_DIR=$TRT_LIBPATH/lib -DTRT_OUT_DIR=`pwd`/out -DCUDA_VERSION=11.0
make -j$(nproc)
```

Note: unsupported GNU version! gcc versions later than 7 are not supported!

~~~
```bash
ln -s /usr/bin/gcc-7 /usr/local/cuda/bin/gcc
ln -s /usr/bin/g++-7 /usr/local/cuda/bin/g++
```
~~~

5. Testing demo

   i. cd ~/TensorRT-7.2.3.4/data/mnist # this is the GA file

   ii. pip install Pillow

   iii. python download_pgms.py

   iv. cd /path_to/TensorRT/build/out # this is the TensorRtOss

   v. ./sample_mnist --datadir ~/TensorRT-7.2.3.4/data/mnist/

Ref：

https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html#installing-tar

https://github.com/NVIDIA/TensorRT/

https://www.codenong.com/cs106336708/

https://segmentfault.com/a/1190000039977778

2. Convert Pytorch Model to ONNX

As mentioned before, nerual machine translation has different inference workflow and model structure from training. Dhe decoder is highly coupled with beam search in the inference. Also, considering that beam search may be relatively unsuitable to be built into tensorRT engine. Thus, we decide to convert the encoder and the decoder to TensorRT engines separately. And the inference steps should be connected with beam search implemented in torch. This will also keep the enc-dec model flexible.

Another thing is that the initial version of docker should be implemented separately, because the generator has to start with a non-cached-mems decoder.

2.1 Export encoder

The inference encoder is the same as it is in the training mode. We can simply call nemo function `nemo_model.export(output=onnx_path)` to convert it into onnx. Dynamic inputs are supported here, including batch_size and sequence length.

2.2 Export decoder

This is the first trick part. We have to modify the original code to make two decoder versions. And reorienting the `forward` function when exporting to onnx.

1. init decoder without cached memories

```
def init_forward(self, decoder_states, decoder_mask, encoder_states, encoder_mask):
    decoder_attn_mask = form_attention_mask(decoder_mask, diagonal=self.diagonal)
    encoder_attn_mask = form_attention_mask(encoder_mask)
    memory_states = decoder_states
    cached_mems_list = [memory_states.unsqueeze(0)]

    for i, layer in enumerate(self.layers):
        decoder_states = layer(decoder_states, decoder_attn_mask, memory_states, encoder_states,
encoder_attn_mask)
        memory_states = decoder_states
        cached_mems_list.append(memory_states.unsqueeze(0))

    cached_mems = torch.cat(cached_mems_list, 0)
    return cached_mems
```

before exporting, you should reorient `decoder.forward = decoder.init_forward` and specify the dynamic axes

```
dynamic_axes={
            'decoder_states':{0:'batch_size'},
            'decoder_mask':{0:'batch_size'},
            'encoder_states':{0:'batch_size',1:'seq_len'},
            'encoder_mask':{0:'batch_size',1:'seq_len'},
            'cached_mems':{1:'batch_size'},
            }
```

2. deocder with cached memories

```
def non_init_forward(self, decoder_states, decoder_mask, encoder_states, encoder_mask,
decoder_mems_list):
     decoder_attn_mask = form_attention_mask(decoder_mask, diagonal=self.diagonal)
     encoder_attn_mask = form_attention_mask(encoder_mask)
     memory_states = torch.cat((decoder_mems_list[0], decoder_states), dim=1)
     cached_mems_list = [memory_states.unsqueeze(0)]

     for i, layer in enumerate(self.layers):
         decoder_states = layer(decoder_states, decoder_attn_mask, memory_states, encoder_states,
encoder_attn_mask)
         memory_states = torch.cat((decoder_mems_list[i+1], decoder_states), dim=1)
         cached_mems_list.append(memory_states.unsqueeze(0))

     cached_mems = torch.cat(cached_mems_list, 0)
     return cached_mems
```

before exporting, you should also reorient `decoder.forward = decoder.non_init_forward` and specify the dynamic axes.

```
dynamic_axes={
            'decoder_states':{0:'batch_size'},
            'decoder_mask':{0:'batch_size'},
            'encoder_states':{0:'batch_size',1:'seq_len'},
            'encoder_mask':{0:'batch_size',1:'seq_len'},
            'decoder_mems':{1:'batch_size',2:'dec_len'},
            'cached_mems':{1:'batch_size',2:'dec_len'}
            }
```

3. Convert ONNX to TensorRT

There are two approaches, `trtexec` and `python API`, to build TensorRT engine from onnx. The latter is recommanded due to its high flexibility.

3.1 trtexec tool

Very easy to use, you can check the config by run `trtexec -v`.

```
trtexec --onnx=${enc_onnx_path} \
    --explicitBatch \
    --minShapes=${enc_min_shape} \
    --optShapes=${enc_opt_shape} \
    --maxShapes=${enc_max_shape} \
    --fp16 \
    --shapes=${enc_inf_shape} \
    --saveEngine=${enc_trt_path}
```

Very easy to test engine performace.

```
trtexec --loadEngine=${enc_trt_path} \
        --shapes=${enc_inf_shape} \
        --iterations=10
```

Add `--dumpProfile` to print network profile.

Note: always use explicit batch instead of implicit batch

3.2 python api

If you are using dynamic axes, please create optimization profile for builder as `profile = builder.create_optimization_profile()`. Also, you should specify min, opt, max dims for every dynamic axes in this profile and the make sure the names match. When building engine in fp16 mode, you need to active `builder.fp16_mode` and set config flag `trt.BuilderFlag.FP16`. Besides, if you want to build an int8 engine, a calib file must be provided.

```
def build_engine(model_file, shapes, max_batch_size, fp16_mode=False, int8_mode=False, calib=""):
    """Takes an ONNX file and creates a TensorRT engine to run inference with"""
    # onnx not support implicit batch and must specify the explicit batch
    explicit_batch = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
    with trt.Builder(TRT_LOGGER) as builder, \
            builder.create_network(explicit_batch) as network, \
            trt.OnnxParser(network, TRT_LOGGER) as parser:

        config = builder.create_builder_config()
        config.max_workspace_size = GiB(8)
        builder.max_batch_size = max_batch_size
        if fp16_mode:
            assert (builder.platform_has_fast_fp16 == True), "not support fp16"
            builder.fp16_mode = True
            config.set_flag(trt.BuilderFlag.FP16)
        if int8_mode:
            assert (builder.platform_has_fast_int8 == True), "not support int8"
            builder.int8_mode = True
            builder.int8_calibrator = calib
            raise NotImplementedError

        print('Loading ONNX file from path {}...'.format(model_file))
        with open(model_file, 'rb') as model:
            if not parser.parse(model.read()):
                print ('ERROR: Failed to parse the ONNX file.')
                for error in range(parser.num_errors):
                    print (parser.get_error(error))
                return None
        print('Complete parsing of ONNX file')

        # dynamic inputs setting
        profile = builder.create_optimization_profile()
        for s in shapes:
            profile.set_shape(s['name'], min=s['min'], opt=s['opt'], max=s['max'])
        config.add_optimization_profile(profile)

        engine = builder.build_engine(network,config)
        return engine
```

Note: increase workspace if you meet memery error when building engine

4. Run TensorRT Inference

The normal way to run tensorRT inference is like

```
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__()
```

1. load an engine and create a context from it
2. specify `context.active_optimization_profile = 0` if using dynamic axes
3. reshape context bindings
4. allocate buffers on CPU and GPU
5. transfer data from CPU to GPU
6. run inference on GPU
7. move results from GPU back to CPU

```
def do_inference(self, src, src_mask, batch_size=1):
        with self.engine.create_execution_context() as context:
            context.active_optimization_profile = 0
            shapes = [src.shape, src_mask.shape]
            # reshape bindings before doing inference
            for i,shape in enumerate(shapes):
                binding_shape = context.get_binding_shape(i)
                if -1 in binding_shape:
                    binding_shape = tuple(shape)
                    context.set_binding_shape(i,(binding_shape))
            inputs, outputs, bindings, stream = self.allocate_buffers(self.engine,context)
            inputs[0].host = src
            inputs[1].host = src_mask

            # 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]
```

However, this mathod is not suitable for generative model such as NMT task, because the data will be transfered between CPU and GPU for many time (according to sequence length) in the generator loops. This will cost much time in transferring. Therefore, we need to find a more efficient way to run inference. For example, run inference entirely on CUDA without transferring.

```
def run_trt_engine(context, engine, tensors):

    bindings = [None]*engine.num_bindings
    for name,tensor in tensors['inputs'].items():
        idx = engine.get_binding_index(name)
        bindings[idx] = tensor.data_ptr()
        if engine.is_shape_binding(idx) and is_shape_dynamic(context.get_shape(idx)):
            context.set_shape_input(idx, tensor)
        elif is_shape_dynamic(engine.get_binding_shape(idx)):
            context.set_binding_shape(idx, tensor.shape)

    for name,tensor in tensors['outputs'].items():
        idx = engine.get_binding_index(name)
        bindings[idx] = tensor.data_ptr()

    context.execute_v2(bindings=bindings)
```

5. Rough Results

After applying inference on test dataset (1,000 texts), we measure the time for pytorch model and tensorrt model on some specific sessions. The time listed below is the average time per batch.

model	preprocess time	encoder time	generator time	postprocess time	total latency
pytorch	0.0014197865752503275s	0.008084586990065873s	0.417277689971029 76s	0.0016448273165151476s	435.9568956270814s
tensorRT	0.0009395092390477657s	0.0028762537082657217s	0.20571816717181354s	0.0017853414667770267s	211.64874472934753s

6. Issues and Bugs

1. pycuda error

   ImportError: libcurand.so.10.0: cannot open shared object file: No such file or directory

   sol：cache issue，you maight have installed it before and it is kept in cached，but it does not match the current cuda version. do `rm -r .cache/pip/`

   ref： https://github.com/inducer/pycuda/issues/204

2. cannot change maxBatch if build engine by trtexec then run in python

   sol: maxBatch cannot be used with ONNX, which only supports explicit batch dimension. use explicit match and use python api to builf engine

3. building engine failed because of residual connection?

   debug: print out verbose info and check network using netron, debub layer by layer

   sol: ignore it for two days and solved

4. cuda exec error when using context.execute_v2(bindings=bindings)

   [TensorRT] ERROR: ../rtSafe/cuda/genericReformat.cu (1294) - Cuda Error in executeMemcpy: 1 (invalid argument)

   sol: maybe due to the environment, try nv docker

5. fp16 bug

   When using fp16 engine, the precision is highly deduced.

   debug: check onnx network using netron and print out network layers and nodes

   ```
   def mark_outputs(network):
   print('network.num_layers:',network.num_layers)
   for i in range(network.num_layers-1):
       layer = network.get_layer(i)
       print(i,layer.name,layer.type)
       continue
       for j in range(layer.num_outputs):
           network.mark_output(layer.get_output(j))
   ```

   sol: this bug happens because trt fuses and optimizes some nodes but cause overflow in fp16 mode. We can use `mark_outputs` function to stop trt optimize these nodes.

   ```
   idx = [400, 680, 960, 1240, 1520] # dec_init
        idx = [210, 493, 776, 1059, 1342] # dec
        for i in idx:
            layer = network.get_layer(i)
            print('layer:',i,layer.name,layer.type,layer.precision)
            for j in range(layer.num_outputs):
                output = layer.get_output(j)
                print('output:',output.name,output.shape)
                network.mark_output(output)
   ```

   ref: https://bbs.cvmart.net/articles/4531