# 使用TensorRT对网络加速及在TX2上部署（caffe）

以下演示使用TensorRT对已经训练好的网格模型进行加速和部署的步骤。

## TensorRT C++ API的使用

### 创建Logger

许多像IRuntime, IBuilder这样的实例在创建时都要使用同一个Logger对象，一个简单的Logger类可以这样定义：

|  |
| --- |
| class Logger : public nvinfer1::ILogger  {      void log(Severity severity, const char\* msg) override      {          // suppress info-level messages          if (severity != Severity::kINFO)              std::cout << msg << std::endl;      }  }; |

### 创建GIE模型并序列化到流中

该步骤使用网络结构文件（.prototxt）、权重文件（.caffemodel）创建Engine对象，并序列化到流中，以备后续使用。

首先使用上面的Logger对象为参数，创建IBuiler对象：

|  |
| --- |
| nvinfer1::IBuilder\* builder = nvinfer1::createInferBuilder(logger); |

创建INetworkDefinition对象：

|  |
| --- |
| nvinfer1::INetworkDefinition\* network = builder->createNetwork(); |

创建ICaffeParser对象并对网络文件进行解析：

|  |
| --- |
| nvcaffeparser1::ICaffeParser\* parser = nvcaffeparser1::createCaffeParser();   const nvcaffeparser1::IBlobNameToTensor\* blobNameToTensor = parser->parse(deployFile.c\_str(), modelFile.c\_str(), \*network, nvinfer1::DataType::kFLOAT); |

设置输出的张量：

|  |
| --- |
| for (auto& s : outputs)          network->markOutput(\*blobNameToTensor->find(s.c\_str())); |

设置IBuilder的相关参数并创建ICudaEngine对象：

|  |
| --- |
| builder->setMaxBatchSize(maxBatchSize);   builder->setMaxWorkspaceSize(1 << 20);   nvinfer1::ICudaEngine\* engine = builder->buildCudaEngine(\*network); |

将engine序列化到流中，后续从流中反序列化得到engine：

|  |
| --- |
| // serialize the engine, then close everything down   gieModelStream = engine->serialize(); |

销毁中间变量：

|  |
| --- |
| network->destroy();  parser->destroy();  engine->destroy();  builder->destroy();  nvcaffeparser1::shutdownProtobufLibrary(); |

### **创建IRuntime对象并反序列化得到ICudaEngine对象**

|  |
| --- |
| nvinfer1::IRuntime\* runtime = nvinfer1::createInferRuntime(logger);  nvinfer1::ICudaEngine\* engine = runtime->deserializeCudaEngine(gieModelStream->data(), gieModelStream->size(), nullptr); |

### **使用ICudaEngine对象创建IExecutionContext**

IExecutionContext用于执行前向推理，使用上面的engine创建：

|  |
| --- |
| nvinfer1::IExecutionContext\* context = engine->createExecutionContext(); |

### **处理均值文件**

由于这里我们用到了均值文件，需要解析均值文件：

|  |
| --- |
| nvcaffeparser1::ICaffeParser\* parser = nvcaffeparser1::createCaffeParser();  nvcaffeparser1::IBinaryProtoBlob\* meanBlob = parser->parseBinaryProto(mean\_file.c\_str());  parser->destroy();  const float\* meanData = reinterpret\_cast<const float\*>(meanBlob->getData()); |

### **前向推理**

读取图像文件，并进行预处理：

|  |
| --- |
| cv::Mat mat = cv::imread(std::string(buf), 0);   if (!mat.data) {         fprintf(stderr, "read image fail: %s\n", buf);        continue;   }   cv::resize(mat, mat, cv::Size(info.width, info.height));   mat.convertTo(mat, CV\_32FC1);  mat = 255 - mat;  float \*data = new float[info.width \* info.height];  const float\* p = (float\*)mat.data;  for (int j = 0; j < info.width \* info.height; ++j)  {         data[j] = p[j] - meanData[j];   } |

在显存在分配内存，将首地址传给主存buffer上， cudaMalloc返回的是一个cudaError\_t类型的错误代码：

|  |
| --- |
| checkCudaErrors(cudaMalloc(&buffers[inputIndex], batchSize \* info.height \* info.width \* sizeof(float)));    checkCudaErrors(cudaMalloc(&buffers[outputIndex], batchSize \* info.output\_size \* sizeof(float))); |

创建流，将图像数据从主存复制到显存中，入队执行，将执行结果从显存复制回主存，同步。

|  |
| --- |
| cudaStream\_t stream;      checkCudaErrors(cudaStreamCreate(&stream));      // DMA the input to the GPU, execute the batch asynchronously, and DMA it back:      checkCudaErrors(cudaMemcpyAsync(buffers[inputIndex], input, batchSize \* info.height \* info.width \* sizeof(float), cudaMemcpyHostToDevice, stream));      context.enqueue(batchSize, buffers, stream, nullptr);      checkCudaErrors(cudaMemcpyAsync(output, buffers[outputIndex], batchSize \* info.output\_size \* sizeof(float), cudaMemcpyDeviceToHost, stream));      cudaStreamSynchronize(stream); |

释放相关资源：

|  |
| --- |
| // release the stream and the buffers      cudaStreamDestroy(stream);      checkCudaErrors(cudaFree(buffers[inputIndex]));      checkCudaErrors(cudaFree(buffers[outputIndex])); |

### **销毁相关对象**

|  |
| --- |
| meanBlob->destroy();      if (gieModelStream) gieModelStream->destroy();      // destroy the engine      context->destroy();      engine->destroy();      runtime->destroy(); |

## 全部代码

### common.h

|  |
| --- |
| #ifndef FBC\_TENSORRT\_TEST\_COMMON\_HPP\_  #define FBC\_TENSORRT\_TEST\_COMMON\_HPP\_  #include <cuda\_runtime.h>  #include <device\_launch\_parameters.h>  #include <NvInfer.h>  #include <iostream>  template< typename T >  static inline int check\_Cuda(T result, const char \* const func, const char \* const file, const int line)  {      if (result) {          fprintf(stderr, "Error CUDA: at %s: %d, error code=%d, func: %s\n", file, line, static\_cast<unsigned int>(result), func);          cudaDeviceReset(); // Make sure we call CUDA Device Reset before exiting          return -1;      }  }  template< typename T >  static inline int check(T result, const char \* const func, const char \* const file, const int line)  {      if (result) {          fprintf(stderr, "Error: at %s: %d, error code=%d, func: %s\n", file, line, static\_cast<unsigned int>(result), func);          return -1;      }  }  #define checkCudaErrors(val) check\_Cuda((val), \_\_FUNCTION\_\_, \_\_FILE\_\_, \_\_LINE\_\_)  #define checkErrors(val) check((val), \_\_FUNCTION\_\_, \_\_FILE\_\_, \_\_LINE\_\_)  #define CHECK(x) {\  if(x) {} \  else { fprintf(stderr, "Check Failed: %s, file: %s, line: %d\n", #x, \_\_FILE\_\_, \_\_LINE\_\_); return -1; } \  }  // Logger for GIE info/warning/errors  class Logger : public nvinfer1::ILogger  {      void log(Severity severity, const char\* msg) override      {          // suppress info-level messages          if (severity != Severity::kINFO)              std::cout << msg << std::endl;      }  };  #endif // FBC\_TENSORRT\_TEST\_COMMON\_HPP\_ |

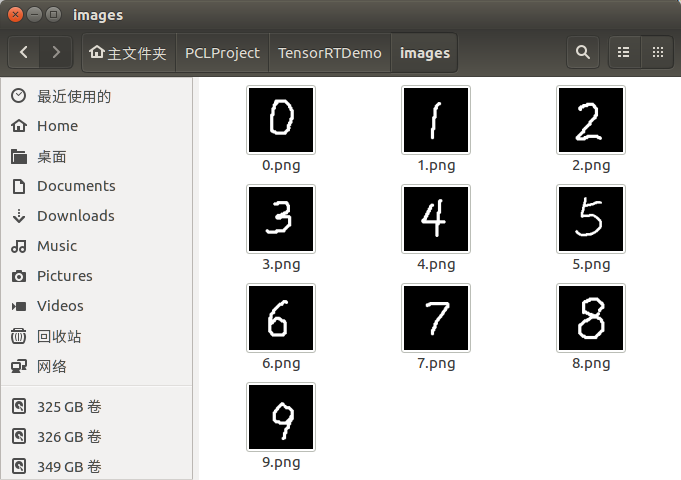
### mnist\_test.cpp

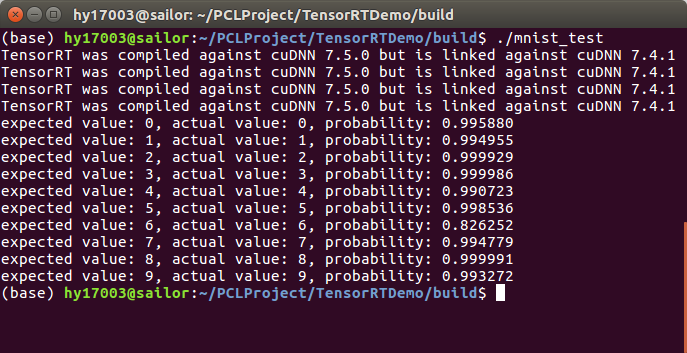
|  |
| --- |
| #include <iostream>  #include <string>  #include <cuda\_runtime\_api.h>  #include <NvInfer.h>  #include <NvCaffeParser.h>  #include <opencv2/opencv.hpp>  #include "common.h"  struct DATA\_INFO  {      int width;      int height;      int output\_size;      std::string input\_blob\_name;      std::string output\_blob\_name;  };  static int caffeToGIEModel(const std::string& deployFile,   // name for caffe prototxt      const std::string& modelFile,   // name for model      const std::vector<std::string>& outputs, // network outputs      unsigned int maxBatchSize,  // batch size - NB must be at least as large as the batch we want to run with)      nvinfer1::IHostMemory \*&gieModelStream, // output buffer for the GIE model      Logger logger)  {      // create the builder      nvinfer1::IBuilder\* builder = nvinfer1::createInferBuilder(logger);      // parse the caffe model to populate the network, then set the outputs      nvinfer1::INetworkDefinition\* network = builder->createNetwork();      nvcaffeparser1::ICaffeParser\* parser = nvcaffeparser1::createCaffeParser();      const nvcaffeparser1::IBlobNameToTensor\* blobNameToTensor = parser->parse(deployFile.c\_str(), modelFile.c\_str(), \*network, nvinfer1::DataType::kFLOAT);      // specify which tensors are outputs      for (auto& s : outputs)          network->markOutput(\*blobNameToTensor->find(s.c\_str()));      // Build the engine      builder->setMaxBatchSize(maxBatchSize);      builder->setMaxWorkspaceSize(1 << 20);      nvinfer1::ICudaEngine\* engine = builder->buildCudaEngine(\*network);      CHECK(engine != nullptr);      // serialize the engine, then close everything down      gieModelStream = engine->serialize();      // we don't need the network any more, and we can destroy the parser      network->destroy();      parser->destroy();      engine->destroy();      builder->destroy();      nvcaffeparser1::shutdownProtobufLibrary(); ///// Note      return 0;  }  static int doInference(nvinfer1::IExecutionContext& context, const float\* input, float\* output, int batchSize, const DATA\_INFO& info)  {      const nvinfer1::ICudaEngine& engine = context.getEngine();      // input and output buffer pointers that we pass to the engine - the engine requires exactly IEngine::getNbBindings(),      // of these, but in this case we know that there is exactly one input and one output.      CHECK(engine.getNbBindings() == 2);      void\* buffers[2];      // In order to bind the buffers, we need to know the names of the input and output tensors.      // note that indices are guaranteed to be less than IEngine::getNbBindings()      int inputIndex = engine.getBindingIndex(info.input\_blob\_name.c\_str()),      outputIndex = engine.getBindingIndex(info.output\_blob\_name.c\_str());      // create GPU buffers and a stream      checkCudaErrors(cudaMalloc(&buffers[inputIndex], batchSize \* info.height \* info.width \* sizeof(float)));      checkCudaErrors(cudaMalloc(&buffers[outputIndex], batchSize \* info.output\_size \* sizeof(float)));      cudaStream\_t stream;      checkCudaErrors(cudaStreamCreate(&stream));      // DMA the input to the GPU, execute the batch asynchronously, and DMA it back:      checkCudaErrors(cudaMemcpyAsync(buffers[inputIndex], input, batchSize \* info.height \* info.width \* sizeof(float), cudaMemcpyHostToDevice, stream));      context.enqueue(batchSize, buffers, stream, nullptr);      checkCudaErrors(cudaMemcpyAsync(output, buffers[outputIndex], batchSize \* info.output\_size \* sizeof(float), cudaMemcpyDeviceToHost, stream));      cudaStreamSynchronize(stream);      // release the stream and the buffers      cudaStreamDestroy(stream);      checkCudaErrors(cudaFree(buffers[inputIndex]));      checkCudaErrors(cudaFree(buffers[outputIndex]));      return 0;  }  int main(int argc, char\* argv[])  {      const DATA\_INFO info{28, 28, 10, "data", "prob"};      const std::string deploy\_file{ "../model/mnist.prototxt" };      const std::string model\_file{ "../model/mnist.caffemodel" };      const std::string mean\_file{ "../model/mnist\_mean.binaryproto" };      const std::vector<std::string> output\_blobs\_name{info.output\_blob\_name};      Logger logger; // multiple instances of IRuntime and/or IBuilder must all use the same logger      // create a GIE model from the caffe model and serialize it to a stream      nvinfer1::IHostMemory\* gieModelStream{ nullptr };      caffeToGIEModel(deploy\_file, model\_file, output\_blobs\_name, 1, gieModelStream, logger);      // parse the mean file and  subtract it from the image      nvcaffeparser1::ICaffeParser\* parser = nvcaffeparser1::createCaffeParser();      nvcaffeparser1::IBinaryProtoBlob\* meanBlob = parser->parseBinaryProto(mean\_file.c\_str());      parser->destroy();      // deserialize the engine      nvinfer1::IRuntime\* runtime = nvinfer1::createInferRuntime(logger);      nvinfer1::ICudaEngine\* engine = runtime->deserializeCudaEngine(gieModelStream->data(), gieModelStream->size(), nullptr);      nvinfer1::IExecutionContext\* context = engine->createExecutionContext();      const float\* meanData = reinterpret\_cast<const float\*>(meanBlob->getData());      for(int i = 0;i<10;i++)      {          char buf[100];          sprintf(buf, "../images/%d.png", i);          cv::Mat mat = cv::imread(std::string(buf), 0);          if (!mat.data) {              fprintf(stderr, "read image fail: %s\n", buf);              continue;          }          cv::resize(mat, mat, cv::Size(info.width, info.height));          mat.convertTo(mat, CV\_32FC1);          mat = 255 - mat;          float \*data = new float[info.width \* info.height];          const float\* p = (float\*)mat.data;          for (int j = 0; j < info.width \* info.height; ++j)          {              data[j] = p[j] - meanData[j];          }          // run inference          float \*prob = new float[info.output\_size];          doInference(\*context, data, prob, 1, info);          float val{ -1.f };          int idx{ -1 };          for (int t = 0; t < info.output\_size; ++t)          {              if (val < prob[t])              {                  val = prob[t];                  idx = t;              }          }          delete[] prob;          delete[] data;          fprintf(stdout, "expected value: %d, actual value: %d, probability: %f\n", i, idx, val);      }      meanBlob->destroy();      if (gieModelStream) gieModelStream->destroy();      // destroy the engine      context->destroy();      engine->destroy();      runtime->destroy();      return 0;  } |

### CMakeLists.txt

|  |
| --- |
| cmake\_minimum\_required(VERSION 2.8)  PROJECT(mnist\_test)  SET(CMAKE\_CXX\_STANDARD 11)  FIND\_PACKAGE(OpenCV REQUIRED)  FIND\_PACKAGE(CUDA REQUIRED)  INCLUDE\_DIRECTORIES(./include  /home/hy17003/TensorRT-5.1.5.0/include  ${CUDA\_INCLUDE\_DIRS}  )  SET(TENSORRT\_LIB\_PATH "/home/hy17003/TensorRT-5.1.5.0/lib")  FILE(GLOB TENSORRT\_LIBS "${TENSORRT\_LIB\_PATH}/\*.so")  AUX\_SOURCE\_DIRECTORY(src SRC\_LIST)  ADD\_EXECUTABLE(${PROJECT\_NAME} ${SRC\_LIST})  TARGET\_LINK\_LIBRARIES(${PROJECT\_NAME}${OpenCV\_LIBS} ${TENSORRT\_LIBS} ${CUDA\_LIBRARIES}) |

### 执行结果





## TX2上部署

首先在TX2上安装jetson-inference， 地址：

<https://github.com/dusty-nv/jetson-inference>

将上面工程复制到TX2上，修改CMakeLists.txt:

|  |
| --- |
| cmake\_minimum\_required(VERSION 2.8)  PROJECT(mnist\_test)  SET(CMAKE\_CXX\_STANDARD 11)  FIND\_PACKAGE(OpenCV REQUIRED)  FIND\_PACKAGE(jetson-utils)  FIND\_PACKAGE(jetson-inference)  find\_package(CUDA)  find\_package(Qt4)  include(${QT\_USE\_FILE})  add\_definitions(${QT\_DEFINITIONS})  INCLUDE\_DIRECTORIES(./include  ${CUDA\_INCLUDE\_DIRS}  )  AUX\_SOURCE\_DIRECTORY(src SRC\_LIST)  ADD\_EXECUTABLE(${PROJECT\_NAME} ${SRC\_LIST})  TARGET\_LINK\_LIBRARIES(${PROJECT\_NAME} ${OpenCV\_LIBS} jetson-inference ${CUDA\_LIBRARIES}) |

编译执行：

