# 使用TensorRT对网络加速(tf)

以下演示使用TensorRT对已经训练好的tensorflow模型进行加速。

## 将tensorflow模型转成uff格式

此处由于写本篇文档时手上没有tensorflow模型，故省略，转换方法可使用tensorRT的python API：uff.from\_tensorflow。

这里我使用tensorRT提供的lenet.uff文件。

## 主要流程

### 2.1 注册tensorflow的输入和输出

使用parser的registerInput和reginsterOutput函数：

|  |
| --- |
| //创建uff解析器  auto parser = createUffParser();  //注册tensorflow输入输出  parser->registerInput("in", Dims3(1, 28, 28), UffInputOrder::kNCHW);  parser->registerOutput("out"); |

### 2.2 载入uff文件并创建engine

|  |
| --- |
| //载入uff并创建engine  std::string fileName = "../model/lenet5.uff";  ICudaEngine\* engine = loadModelAndCreateEngine(fileName.c\_str(), maxBatchSize, parser); |

其中loadModelAndCreateEngine定义如下：

|  |
| --- |
| ICudaEngine\* loadModelAndCreateEngine(const char\* uffFile, int maxBatchSize,  IUffParser\* parser)  {  IBuilder\* builder = createInferBuilder(logger);  assert(builder != nullptr);  INetworkDefinition\* network = builder->createNetwork();  if (!parser->parse(uffFile, \*network, nvinfer1::DataType::kFLOAT))  {  std::cout << "Failure while parsing UFF file" << std::endl;  return nullptr;  }  /\* we create the engine \*/  builder->setMaxBatchSize(maxBatchSize);  builder->setMaxWorkspaceSize(MAX\_WORKSPACE);  builder->setFp16Mode(true);  builder->setInt8Mode(false);  ICudaEngine\* engine = builder->buildCudaEngine(\*network);  if (!engine)  {  std::cout << "Unable to create engine" << std::endl;  return nullptr;  }  network->destroy();  builder->destroy();  return engine;  } |

### 2.3 创建IExecutionContext

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

### 2.4 前向推理

|  |
| --- |
| 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;  } |

## 代码

### 3.1 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\_ |

### 3.2 uff\_mnist.cpp

|  |
| --- |
| #include <algorithm>  #include <chrono>  #include <cstdlib>  #include <cuda\_runtime\_api.h>  #include <fstream>  #include <iostream>  #include <string>  #include <sys/stat.h>  #include <unordered\_map>  #include <cassert>  #include <vector>  #include "NvInfer.h"  #include "NvUffParser.h"  #include <opencv2/opencv.hpp>  #include "NvUtils.h"  #include "common.h"  using namespace nvuffparser;  using namespace nvinfer1;  struct DATA\_INFO  {      int width;      int height;      int output\_size;      std::string input\_blob\_name;      std::string output\_blob\_name;  };  Logger logger;  const DATA\_INFO info{28, 28, 10, "in", "out"};  #define MAX\_WORKSPACE (1 << 30)  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;  }  ICudaEngine\* loadModelAndCreateEngine(const char\* uffFile, int maxBatchSize,  IUffParser\* parser)  {  IBuilder\* builder = createInferBuilder(logger);  assert(builder != nullptr);  INetworkDefinition\* network = builder->createNetwork();  if (!parser->parse(uffFile, \*network, nvinfer1::DataType::kFLOAT))  {  std::cout << "Failure while parsing UFF file" << std::endl;  return nullptr;  }  /\* we create the engine \*/  builder->setMaxBatchSize(maxBatchSize);  builder->setMaxWorkspaceSize(MAX\_WORKSPACE);  builder->setFp16Mode(true);  builder->setInt8Mode(false);  ICudaEngine\* engine = builder->buildCudaEngine(\*network);  if (!engine)  {  std::cout << "Unable to create engine" << std::endl;  return nullptr;  }  network->destroy();  builder->destroy();  return engine;  }  int main(int argc, char\*\* argv)  {  int maxBatchSize = 1;  //创建uff解析器  auto parser = createUffParser();  //注册tensorflow输入输出  parser->registerInput("in", Dims3(1, 28, 28), UffInputOrder::kNCHW);  parser->registerOutput("out");  //载入uff并创建engine  std::string fileName = "../model/lenet5.uff";  ICudaEngine\* engine = loadModelAndCreateEngine(fileName.c\_str(), maxBatchSize, parser);  IExecutionContext\* context = engine->createExecutionContext();  if (!engine)  {  std::cout << "Model load failed" << std::endl;  }  parser->destroy();  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;          }  //预处理          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] = 1.0 - float(p[j]) / 255.0;          }          // run inference          float \*out = new float[info.output\_size];          doInference(\*context, data, out, 1, info);          float val{ -1.f };          int idx{ -1 };          for (int t = 0; t < info.output\_size; ++t)          {              if (val < out[t])              {                  val = out[t];                  idx = t;              }          }          delete[] out;          delete[] data;          fprintf(stdout, "expected value: %d, actual value: %d, probability: %f\n", i, idx, val);      }  engine->destroy();  shutdownProtobufLibrary();  return 0;  } |

### 3.3 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}) |