TensorRT支持网络自定义层的方法

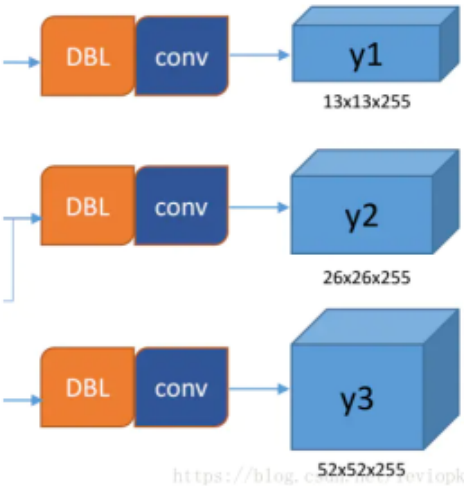
yolov3算法背景

最近研究TRT的自定义层的实现，尝试使用自定义的yolo层，能够省略yolo层后处理的部分代码，实际上就是将Yolo层的多尺度特征输出数据做了nms以及得分阈值过滤，将后处理的大部分工作以网络自定义层的方式，

集成到网络层中，由TensorRT推理引擎处理该自定义层的输入输出，用户直接拿到的网络输出层数据就是yolo层的输出，要了解yolo层自定义的方法，先要了解yolo层到底做了什么工作。我们知道yolov3的框架是一种多尺度

特征预测，对于3 x 416 x416的图像来说，分成大3x52x52，中3x26x26,小3x13x13三种gird（cell）网格，各个尺度下每个Gird都有三个anechor预测框，所谓的anechor就是一种一定面积以及长宽比例的滑动窗口，这个窗口

根据尺寸先验经验对预测的对象范围进行约束，从而实现模型的快速收敛。网络在3个尺度上检测，输出张量如下：



而yolo层则时将三种尺度的张量结果作为网络输入，对每个gird的预测结果做过滤，最后得出最佳score的预测框，因此总的预测框数量为，13x13x3+26x26x3+52x52x3 = 10647个。最终yolo层的输出应该是box框，类别id，得分组成（通过参考开源工程tensorRTWrapper），

下面移植插件的步骤也大多数来源该开源工程。

TRT自定义算子Plugin的实现方法

TensorRT的自定义层机制是有两个方法的，一种基于基类IPlugin，另一种是基于基类IPluginV2，从字面意思上来看IPluginV2就是最新版本。
　　1、IPlugin类的方法，是通过自己编写IPlugin的派生类IPluginExt和nvinfer1::IPluginFactory类以及nvcaffeparser1::IPluginFactoryExt类来实现自定义层的编辑和使用的。就是通过IPluginFactory类将IPluginExt类定义的Plugin实例化，然后在TensorRT中使用。在tensorRTWrapper中对应的是如下这段代码：

IPluginExt派生自定义插件，类中实现必要的虚函数以及重载接口函数

```
namespace nvinfer1
{
    class YoloLayerPlugin: public IPluginExt
    {
    public:
        explicit YoloLayerPlugin(const int cudaThread = 512);
        YoloLayerPlugin(const void* data, size_t length);



        ~YoloLayerPlugin();

        int getNbOutputs() const override
        {
            return 1;
        }

        Dims getOutputDimensions(int index, const Dims* inputs, int nbInputDims) override;

        bool supportsFormat(DataType type, PluginFormat format) const override {
            return type == DataType::kFLOAT && format == PluginFormat::kNCHW;
        }

        void configureWithFormat(const Dims* inputDims, int nbInputs, const Dims* outputDims, int nbOutputs,
DataType type, PluginFormat format, int maxBatchSize) override {};

        int initialize() override;

        virtual void terminate() override {};

        virtual size_t getWorkspaceSize(int maxBatchSize) const override { return 0;}

        virtual int enqueue(int batchSize, const void*const * inputs, void** outputs, void* workspace,
cudaStream_t stream) override;

        virtual size_t getSerializationSize() override;

        virtual void serialize(void* buffer) override;

        void forwardGpu(const float *const * inputs,float * output, cudaStream_t stream,int batchSize = 1);

        void forwardCpu(const float *const * inputs,float * output, cudaStream_t stream,int batchSize = 1);

    private:
        int mClassCount;
        int mKernelCount;
        std::vector<Yolo::YoloKernel> mYoloKernel;
        int mThreadCount;

        //cpu
        void* mInputBuffer  {nullptr};
        void* mOutputBuffer {nullptr};
    };
};
```

PluginFactory 负责将自定义的Plugin实例化，通过接口parser->setPluginFactory(pluginFactory)或者parser->setPluginFactoryExt(pluginFactory)；

```
    class PluginFactory : public nvinfer1::IPluginFactory, public nvcaffeparser1::IPluginFactoryExt
    {
        public:
            inline bool isLeakyRelu(const char* layerName)
            {
                return std::regex_match(layerName , std::regex(R"(layer(\d*)-act)"));
            }
```

```
            inline bool isUpsample(const char* layerName)
            {
                return std::regex_match(layerName , std::regex(R"(layer(\d*)-upsample)"));
            }

            inline bool isYolo(const char* layerName)
            {
                return strcmp(layerName,"yolo-det") == 0;
            }

            virtual nvinfer1::IPlugin* createPlugin(const char* layerName, const nvinfer1::Weights* weights,
int nbWeights) override
            {
                assert(isPlugin(layerName));

                if (isUpsample(layerName))
                {
                    assert(nbWeights == 0 && weights == nullptr);
                    mPluginUpsample.emplace_back(std::unique_ptr<UpsampleLayerPlugin>(new UpsampleLayerPlugin
(UPSAMPLE_SCALE,CUDA_THREAD_NUM)));
                    return mPluginUpsample.back().get();
                }
                else if (isYolo(layerName))
                {
                    assert(nbWeights == 0 && weights == nullptr && mPluginYolo.get() ==  nullptr);
                    mPluginYolo.reset(new YoloLayerPlugin(CUDA_THREAD_NUM));
                    return mPluginYolo.get();
                }
                else
                {
                assert(0);
                return nullptr;
                }
            }

        nvinfer1::IPlugin* createPlugin(const char* layerName, const void* serialData, size_t serialLength)
override
        {
            assert(isPlugin(layerName));

            if (isUpsample(layerName))
            {
                mPluginUpsample.emplace_back(std::unique_ptr<UpsampleLayerPlugin>(new UpsampleLayerPlugin
(serialData, serialLength)));
                return mPluginUpsample.back().get();
            }
            else if (isYolo(layerName))
            {
                assert(mPluginYolo.get() ==  nullptr);
                mPluginYolo.reset(new YoloLayerPlugin(serialData, serialLength));
                return mPluginYolo.get();
            }
            else
            {
                assert(0);
                return nullptr;
            }
        }


        bool isPlugin(const char* name) override
        {
            return isPluginExt(name);
        }

        bool isPluginExt(const char* name) override
        {
            //std::cout << "check plugin " << name  << isYolo(name)<< std::endl;
            return isUpsample(name) || isYolo(name);
        }
```

```
        // The application has to destroy the plugin when it knows it's safe to do so.
        void destroyPlugin()
        {
            for (auto& item : mPluginLeakyRelu)
                item.reset();

            for (auto& item : mPluginUpsample)
                item.reset();

            mPluginYolo.reset();
        }

        void (*nvPluginDeleter)(INvPlugin*){[](INvPlugin* ptr) { if(ptr) ptr->destroy(); }};

        std::vector<std::unique_ptr<INvPlugin,void (*)(INvPlugin*)>> mPluginLeakyRelu{};
        std::vector<std::unique_ptr<UpsampleLayerPlugin>> mPluginUpsample{};
        std::unique_ptr<YoloLayerPlugin> mPluginYolo {nullptr};
    };
```

IPluginV2类的方法，是关于自定义层的另一种方法，这种方法是调用已经被注册的plugin的方法，这种方法不依赖nvinfer1::IPluginFactory去实例化，而是使用nvcaffeparser1::IPluginFactoryV2 和 IPluginCreator代替，这种方法对应trt官方插件代码中的中的如下代码：

Upsample类继承IPluginV2Ext，类中同样要实现必要的重载接口和虚函数，其实例化是通过继承BaseCreator类的UpsamplePluginCreator类实现，其构造函数实例化该自定义层

```
namespace nvinfer1
{
    namespace plugin
    {
        class Upsample : public IPluginV2Ext
        {
        public:
            //constructor
            Upsample(int scale);



            //cuda
            Upsample(const void* data, size_t length);

            ~Upsample() override = default;

            int getNbOutputs() const override;

            Dims getOutputDimensions(int index,const Dims* inputs,int nbInputDims) override;

            int initialize() override;

            void terminate() override;

            void destroy() override;

            size_t getWorkspaceSize(int maxBatchSize) const override;

            //cuda
            int enqueue(
                    int batchSize, const void* const* inputs, void** outputs, void* workspace,cudaStream_t
stream) override;

            size_t getSerializationSize() const override;

            void serialize(void* buffer) const override;

            bool supportsFormat(DataType type,PluginFormat format) const override;

            const char* getPluginType() const override;

            const char* getPluginVersion() const override;

            IPluginV2Ext* clone() const override;
```

```
            void setPluginNamespace(const char* pluginNamespace) override;

            const char* getPluginNamespace() const override;

            DataType getOutputDataType(int index, const nvinfer1::DataType* inputTypes, int nbInputs) const
override;

            bool isOutputBroadcastAcrossBatch(int outputIndex, const bool* inputIsBroadcasted, int nbInputs)
const override;

            bool canBroadcastInputAcrossBatch(int inputIndex) const override;

            void attachToContext(
                    cudnnContext* cudnnContext, cublasContext* cublasContext, IGpuAllocator* gpuAllocator)
override;

            void configurePlugin(const Dims* inputDims, int nbInputs, const Dims* outputDims, int nbOutputs,
                                 const DataType* inputTypes, const DataType* outputTypes, const bool*
inputIsBroadcast,
                                 const bool* outputIsBroadcast, PluginFormat floatFormat, int maxBatchSize)
override;

            void detachFromContext() override;

        private:
            int mScale;
            Dims mInputDims;
            Dims mOutputDims;
            const char* mPluginNamespace;
        };

        class UpsamplePluginCreator : public BaseCreator
        {
        public:
            UpsamplePluginCreator();

            ~UpsamplePluginCreator() override = default;

            const char* getPluginName() const override;

            const char* getPluginVersion() const override;

            const PluginFieldCollection* getFieldNames() override;

            IPluginV2Ext* createPlugin(const char* name, const PluginFieldCollection* fc) override;

            IPluginV2Ext* deserializePlugin(const char* name, const void* serialData, size_t serialLength)
override;

        private:
            static PluginFieldCollection mFC;
            int mScale;
            static std::vector<PluginField> mPluginAttributes;
        };
    }
}
```

yolo自定义层的移植步骤

移植yolo自定义层使用的是V2的方法，按照IPluginV2Ext类实现相应接口，按照BaseCreator实例化，步骤如下

生成trt引擎

1、在prototxt中最后添加自定义网络层yolo-det,添加如下：

```
layer {
    bottom: "layer82-conv"
    bottom: "layer94-conv"
    bottom: "layer106-conv"
    top: "yolo-det"
    name: "yolo-det"
    type: "yolo"
}
```

2、模型转换工具修改网络的输出结构：

```
#output=layer82-conv
#output2=layer94-conv
#output3=layer106-conv
output=yolo-det

./caffe2trt  \
--model=${model} \
--deploy=${deploy} \
--engine=${engine} \
--input=${input} \
--output=${output} \
#--output=${output2}
#--output=${output3}
--batch=${batch} \
--dataType=${dataType} \
--dataDir=${dataDir} \
--calibBatchSize=${calibBatchSize} \
--calibMaxBatches=${calibMaxBatches} \
--calibFirstBatch=${calibFirstBatch} \
--device=0 \
--DLACore=-1
```

3、由于是将caffe模型转换为trt引擎模型，因此需要caffe解析器添加队自定义层的的支持：

```
const IBlobNameToTensor* CaffeParser::parse(INetworkDefinition& network,
                                            DataType weightType,
                                            bool hasModel)
{
......
        else if (layerMsg.type() == "yolo")
            {
                pluginName = "Yololayer_TRT";
                f = parseyoloParam(layerMsg, weights, *mBlobNameToTensor);
            }
......
......
}
```

parseyoloParam函数参照其它层的写法实现即可，至此，应该能正常实现模型转换了，得到engine trt模型文件

遇到的问题

1、插件名称对应的插件在注册库找不到--------插件名称空间设置问题，修改createPlugin接口以及deserializePlugin接口：

```
IPluginV2Ext* YololayerPluginCreator::createPlugin(const char *name, const PluginFieldCollection *fc)
{
    //return new YoloLayer(512);
    YoloLayer *obj = new YoloLayer(512);
    obj->setPluginNamespace(mNamespace.c_str());
    return obj;
};



IPluginV2Ext* YololayerPluginCreator::deserializePlugin(const char *name, const void *serialData, size_t
serialLength)
{
    //return new YoloLayer(serialData,serialLength);
    YoloLayer *obj = new YoloLayer(512);
    obj->setPluginNamespace(mNamespace.c_str());
    return obj;
};
```

2、引擎串化的过程中奔溃，clone接口未正常实现：

```
IPluginV2Ext* YoloLayer::clone() const
{
    return new YoloLayer(*this);
};
```