TensorRT中的自定义层

NVIDIA TensorRT 支持多种类型的层，其功能不断扩展；但是，在某些情况下，支持的层不能满足模型的特定需求。

您可以通过实现自定义层（通常称为插件）来扩展 TensorRT。

9.1. Adding Custom Layers Using The C++ API

您可以通过从 TensorRT 的插件基类之一派生来实现自定义层。

Table 3. Base classes, ordered from least expressive to most expressive

	Introduced in TensorRT version?	Mixed input/output formats/types	Dynamic shapes?	Supports implicit/explicit batch mode?
`IPluginV2Ext`	5.1	Limited	No	Implicit batch mode only
`IPluginV2IOExt`	6.0.1	General	No	Implicit batch mode only
`IPluginV2DynamicExt`	6.0.1	General	Yes	Explicit batch mode only

为了在网络中使用插件，您必须首先将其注册到 TensorRT 的`PluginRegistry`（C++、Python）。不是直接注册插件，而是为插件注册一个工厂类的实例，派生自`PluginCreator`（C++，Python）。插件创建者类还提供有关插件的其他信息：它的名称、版本和插件字段参数。

您必须从插件的基类之一派生插件类。在支持具有不同类型/格式的输入/输出或具有动态形状的网络方面，它们具有不同的表达能力。下表总结了基类，按从最不具表现力到最具表现力的顺序排列。

注意：如果插件用于一般用途，请提供 FP32 实现，以使其能够在任何网络上正常运行。

TensorRT 提供了一个宏`REGISTER_TENSORRT_PLUGIN`，它将插件创建者静态注册到注册表中。

TensorRT 库包含可以加载到您的应用程序中的插件。有关开源插件的列表，请参阅`GitHub：TensorRT`插件。

注意：

- 要在应用程序中使用 TensorRT 插件，必须加载`libnvinfer_plugin.so`库，并且必须通过在应用程序代码中调用`initLibNvInferPlugins`来注册所有插件。
- 如果您有自己的插件库，则可以包含一个类似的入口点，以便在唯一命名空间下的注册表中注册所有插件。这确保了在构建期间跨不同插件库的插件名称没有冲突。

有关这些插件的更多信息，请参阅NvInferPlugin.h文件以供参考。

调用`IPluginCreator::createPlugin()`返回`IPluginV2`类型的插件对象。您可以使用`addPluginV2()`将插件添加到 TensorRT 网络，该插件使用给定插件创建网络层。

例如，您可以将插件层添加到您的网络，如下所示：

```
// Look up the plugin in the registry
auto creator = getPluginRegistry()->getPluginCreator(pluginName, pluginVersion);
const PluginFieldCollection* pluginFC = creator->getFieldNames();
// Populate the fields parameters for the plugin layer
PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
// Create the plugin object using the layerName and the plugin meta data
IPluginV2 *pluginObj = creator->createPlugin(layerName, pluginData);
// Add the plugin to the TensorRT network
auto layer = network.addPluginV2(&inputs[0], int(inputs.size()), pluginObj);
… (build rest of the network and serialize engine)
// Destroy the plugin object
pluginObj->destroy()
… (destroy network, engine, builder)
… (free allocated pluginData)
```

注意：

- `pluginData` 必须在传递给 `createPlugin` 之前在堆上分配 `PluginField` 条目。
- 前面描述的 `createPlugin` 方法在堆上创建一个新的插件对象并返回一个指向它的指针。如前所示，确保销毁 `pluginObj` 以避免内存泄漏。

`IPluginV2` 类型插件的插件类型、插件版本和命名空间（如果存在）。在反序列化期间，TensorRT 从插件注册表中查找插件创建者并调用 `IPluginCreator::deserializePlugin()`。在反序列化过程中创建的插件对象由 TensorRT 引擎通过调用 `IPluginV2::destroy()` 方法在内部销毁。

`IPluginV2` 类型插件的插件类型、插件版本和命名空间（如果存在），在反序列化期间，TensorRT 从插件注册表中查找插件创建者并调用 `IPluginCreator::deserializePlugin()`。在反序列化期间创建的插件对象由 TensorRT 引擎通过调用 `IPluginV2::destroy()` 方法在内部销毁。

9.1.1. Example: Adding A Custom Layer With Dynamic Shape Support Using C++

要支持动态形状，您的插件必须从 `IPluginV2DynamicExt` 派生。

关于这个任务

`BarPlugin` 是一个有两个输入和两个输出的插件，其中：
- 第一个输出是第二个输入的拷贝
- 第二个输出是两个输入的串联，沿着第一个维度，所有类型/格式必须相同并且是线性格式

BarPlugin需要按如下方式派生：

```
class BarPlugin : public IPluginV2DynamicExt
{
    ...override virtual methods inherited from IPluginV2DynamicExt.
};
```

继承的方法都是纯虚方法，所以如果你忘记了，编译器会提醒你。

受动态形状影响的四种方法是：

- 获取输出维度
- 支持格式组合

- 配置插件
- 队列

`getOutputDimensions` 的覆盖根据输入维度返回输出维度的符号表达式。您可以使用传递给 `getOutputDimensions` 的 `IExprBuilder` 从输入的表达式构建表达式。在示例中，不必为案例 1 构建新表达式，因为第二个输出的维度与第一个输入的维度相同。

```
DimsExprs BarPlugin::getOutputDimensions(int outputIndex,
    const DimsExprs* inputs, int nbInputs,
    IExprBuilder& exprBuilder)
{
    switch (outputIndex)
    {
    case 0:
    {
        // First dimension of output is sum of input
        // first dimensions.
        DimsExprs output(inputs[0]);
        output.d[0] =
            exprBuilder.operation(DimensionOperation::kSUM,
                inputs[0].d[0], inputs[1].d[0]);
       return output;
    }
    case 1:
        return inputs[0];
    default:
         throw std::invalid_argument("invalid output");
}
```

`supportsFormatCombination` 的覆盖必须指示是否允许格式组合。接口将输入/输出统一索引为“`connections`”，从第一个输入的 0 开始，然后依次为其余输入，然后为输出编号。在示例中，输入是 `connections` 0 和 1，输出是 `connections` 2 和 3。

TensorRT 使用 `supportsFormatCombination` 来询问给定的格式/类型组合是否适用于连接，给定的格式/类型用于索引较少的连接。因此，覆盖可以假设较少索引的连接已经过审查，并专注于与索引 `pos` 的连接。

```
bool BarPlugin::supportsFormatCombination(int pos, const PluginTensorDesc*
inOut, int nbInputs, int nbOutputs) override
{
    assert(0 <= pos && pos < 4);
    const auto* in = inOut;
    const auto* out = inOut + nbInputs;
    switch (pos)
    {
    case 0: return in[0].format == TensorFormat::kLINEAR;
    case 1: return in[1].type == in[0].type &&
                   in[0].format == TensorFormat::kLINEAR;
    case 2: return out[0].type == in[0].type &&
                   out[0].format == TensorFormat::kLINEAR;
    case 3: return out[1].type == in[0].type &&
                   out[1].format == TensorFormat::kLINEAR;
    }
    throw std::invalid_argument("invalid connection number");
```

```
}
```

这里的局部变量 `in` 和 `out` 允许通过输入或输出编号而不是连接编号检查 `inOut` 。

重要提示：覆盖检查索引小于 `pos` 的连接的格式/类型，但绝不能检查索引大于 `pos` 的连接的格式/类型。该示例使用case 3来检查连接 3 和连接 0，而不是使用 case 0来检查连接 0 和连接 3。

TensorRT 使用 `configurePlugin` 在运行时设置插件。这个插件不需要 `configurePlugin` 来做任何事情，所以它是一个空操作：

```
void BarPlugin::configurePlugin(
    const DynamicPluginTensorDesc* in, int nbInputs,
    const DynamicPluginTensorDesc* out, int nbOutputs) override
{
}
```

如果插件需要知道它可能遇到的最小或最大尺寸，它可以检查字段 `DynamicPluginTensorDesc::min` 或 `DynamicPluginTensorDesc::max` 的任何输入或输出。格式和构建时维度信息可以在 `DynamicPluginTensorDesc::desc` 中找到。任何运行时维度都显示为 `-1` 。实际维度提供给 `BarPlugin::enqueue` 。

最后，重写 `BarPlugin::enqueue` 必须完成这项工作。由于形状是动态的，因此 `enqueue` 会收到一个 `PluginTensorDesc` ，它描述了每个输入和输出的实际尺寸、类型和格式。

9.1.2. Example: Adding A Custom Layer With INT8 I/O Support Using C++

PoolPlugin是一个插件，用于演示如何为自定义池层扩展 INT8 I/O。推导如下：

```
class PoolPlugin : public IPluginV2IOExt
{
    ...override virtual methods inherited from IPluginV2IOExt.
};
```

大多数纯虚方法对插件来说都是通用的。影响INT8 I/O的主要方法有：

- 支持格式组合
- 配置插件
- enqueue

`supportsFormatCombination` 的覆盖必须指示允许哪个 INT8 I/O 组合。此接口的用法类似于示例：使用 C++ 添加具有动态形状支持的自定义层。在本例中，支持的 I/O 张量格式为线性 `CHW` ，数据类型为 `FP32` 、 `FP16` 或 `INT8` ，但 I/O 张量必须具有相同的数据类型。

```
bool PoolPlugin::supportsFormatCombination(int pos, const PluginTensorDesc*
inOut, int nbInputs, int nbOutputs) const override
{
    assert(nbInputs == 1 && nbOutputs == 1 && pos < nbInputs + nbOutputs);
    bool condition = inOut[pos].format == TensorFormat::kLINEAR;
    condition &= ((inOut[pos].type == DataType::kFLOAT) ||
                  (inOut[pos].type == DataType::kHALF) ||
                  (inOut[pos].type == DataType::kINT8));
    condition &= inOut[pos].type == inOut[0].type;
    return condition;
}
```

重要的：

- 如果 INT8 校准必须与具有 INT8 I/O 插件的网络一起使用，则该插件必须支持 FP32 I/O，因为它被 FP32 校准图使用。
- 如果不支持 FP32 I/O 变体或未使用 INT8 校准，则必须明确设置所有必需的 INT8 I/O 张量尺度。
- 校准无法确定插件内部张量的动态范围。对量化数据进行操作的插件必须为内部张量计算自己的动态范围。

TensorRT 调用`configurePlugin`方法通过`PluginTensorDesc`将信息传递给插件，这些信息存储为成员变量，序列化和反序列化。

```
void PoolPlugin::configurePlugin(const PluginTensorDesc* in, int nbInput, const
PluginTensorDesc* out, int nbOutput)
{
    ...
    mPoolingParams.mC = mInputDims.d[0];
    mPoolingParams.mH = mInputDims.d[1];
    mPoolingParams.mW = mInputDims.d[2];
    mPoolingParams.mP = mOutputDims.d[1];
    mPoolingParams.mQ = mOutputDims.d[2];
    mInHostScale = in[0].scale >= 0.0f ? in[0].scale : -1.0f;
    mOutHostScale = out[0].scale >= 0.0f ? out[0].scale : -1.0f;
}
```

每个张量的 INT8 I/O 比例可以从`PluginTensorDesc::scale`获得。
最后，重写`UffPoolPluginV2::enqueue`必须完成这项工作。它包括一组核心算法，可在运行时通过使用实际批量大小、输入、输出、cuDNN 流和配置的信息来执行自定义层。

```
int PoolPlugin::enqueue(int batchSize, const void* const* inputs, void**
outputs, void* workspace, cudaStream_t stream)
{
    ...
    CHECK(cudnnPoolingForward(mCudnn, mPoolingDesc, &kONE, mSrcDescriptor,
input, &kZERO, mDstDescriptor, output));
    ...
    return 0;
}
```

9.2. Adding Custom Layers Using The Python API

尽管 C++ API 是实现自定义层的首选语言，但由于可以访问 CUDA 和 cuDNN 等库，您还可以在 Python 应用程序中使用自定义层。

您可以使用 C++ API 创建自定义层，在 Python 中使用`pybind11`打包层，然后将插件加载到 Python 应用程序中。有关更多信息，请参阅在 Python 中创建网络定义。

相同的自定义层实现可用于 C++ 和 Python。

9.2.1. Example: Adding A Custom Layer To A TensorRT Network Using Python

可以使用插件节点将自定义层添加到 Python 中的任何 TensorRT 网络。

Python API 有一个名为`add_plugin_v2`的函数，可让您将插件节点添加到网络。以下示例说明了这一点。它创建了一个简单的TensorRT网络，并通过查找TensorRT插件注册表来添加一个 Leaky ReLU 插件节点。

```
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger()

trt.init_libnvinfer_plugins(TRT_LOGGER, '')
PLUGIN_CREATORS = trt.get_plugin_registry().plugin_creator_list

def get_trt_plugin(plugin_name):
        plugin = None
        for plugin_creator in PLUGIN_CREATORS:
            if plugin_creator.name == plugin_name:
                lrelu_slope_field = trt.PluginField("neg_slope", np.array([0.1],
dtype=np.float32), trt.PluginFieldType.FLOAT32)
                field_collection =
trt.PluginFieldCollection([lrelu_slope_field])
                plugin = plugin_creator.create_plugin(name=plugin_name,
field_collection=field_collection)
        return plugin

def main():
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network()
    config = builder.create_builder_config()
    config.max_workspace_size = 2**20
    input_layer = network.add_input(name="input_layer", dtype=trt.float32,
shape=(1, 1))
    lrelu = network.add_plugin_v2(inputs=[input_layer],
plugin=get_trt_plugin("LReLU_TRT"))
    lrelu.get_output(0).name = "outputs"
    network.mark_output(lrelu.get_output(0))
```

9.3. Using Custom Layers When Importing A Model With A Parser

ONNX 解析器会自动尝试将无法识别的节点作为插件导入。如果在插件注册表中找到与节点具有相同op_type的插件，则解析器将节点的属性作为插件字段参数转发给插件创建者，以创建插件。默认情况下，解析器使用“1”作为插件版本，“”作为插件命名空间。可以通过在相应的 ONNX 节点中设置`plugin_version`或`plugin_namespace`字符串属性来覆盖此行为。

在某些情况下，您可能希望在将 ONNX 图导入 TensorRT 之前对其进行修改。例如，用插件节点替换一组操作。为此，您可以使用ONNX GraphSurgeon 实用程序。有关如何使用 ONNX-GraphSurgeon 替换子图的详细信息，请参阅此示例。

有关更多示例，请参阅onnx_packnet示例。

9.4. Plugin API Description

所有新插件都应从`IPluginCreator`和使用 C++ API 添加自定义层中描述的插件基类之一派生类。此外，新插件还应调用`REGISTER_TENSORRT_PLUGIN(...)`宏以将插件注册到 TensorRT 插件注册表或创建等效于`initLibNvInferPlugins()`的init函数。

9.4.1. Migrating Plugins From TensorRT 6.x Or 7.x To TensorRT 8.x.x

`IPluginV2`和`IPluginV2Ext`以分别向后兼容 TensorRT 5.1 和 6.0.x。但是，新插件应针对`IPluginV2DynamicExt`或`IPluginV2IOExt`接口，而旧插件应重构以使用这些接口。

`IPluginV2DynamicExt`中的新特性如下：

```
virtual DimsExprs getOutputDimensions(int outputIndex, const DimsExprs* inputs,
int nbInputs, IExprBuilder& exprBuilder) = 0;

virtual bool supportsFormatCombination(int pos, const PluginTensorDesc* inOut,
int nbInputs, int nbOutputs) = 0;

virtual void configurePlugin(const DynamicPluginTensorDesc* in, int nbInputs,
const DynamicPluginTensorDesc* out, int nbOutputs) = 0;

virtual size_t getWorkspaceSize(const PluginTensorDesc* inputs, int nbInputs,
const PluginTensorDesc* outputs, int nbOutputs) const = 0;

virtual int enqueue(const PluginTensorDesc* inputDesc, const PluginTensorDesc*
outputDesc, const void* const* inputs, void* const* outputs, void* workspace,
cudaStream_t stream) = 0;
```

`IPluginV2IOExt`中的新特性如下：

```
virtual void configurePlugin(const PluginTensorDesc* in, int nbInput, const
PluginTensorDesc* out, int nbOutput) = 0;

virtual bool supportsFormatCombination(int pos, const PluginTensorDesc* inOut,
int nbInputs, int nbOutputs) const = 0;
```

迁移到`IPluginV2DynamicExt`或`IPluginV2IOExt`的指南：

- `getOutputDimensions`实现给定输入的输出张量维度的表达式。
- `supportsFormatCombination`检查插件是否支持指定输入/输出的格式和数据类型。

- `configurePlugin` 模仿 `IPluginV2Ext` 中等效的 `configurePlugin` 的行为，但接受张量描述符。
- `getWorkspaceSize` 和 `enqueue` 模仿 `IPluginV2Ext` 中等效 API 的行为，但接受张量描述符。

更多详细信息，请参阅IPluginV2 API 说明中的 API 说明。

9.4.2. IPluginV2 API Description

以下部分介绍 `IPluginV2` 类的功能。要将插件层连接到相邻层并设置输入和输出数据结构，构建器通过调用以下插件方法检查输出的数量及其维度。

`getNbOutputs`

用于指定输出张量的数量。

`getOutputDimensions`

用于将输出的维度指定为输入维度的函数。

`supportsFormat`

用于检查插件是否支持给定的数据格式。

`getOutputDataType`

用于获取给定索引处输出的数据类型。返回的数据类型必须具有插件支持的格式。

插件层可以支持四种数据格式，例如：

- NCHW单精度 (FP32)、半精度 (FP16) 和整型 (INT32) 张量
- NC / 2HW2和NHWC8半精度 (FP16) 张量

格式由 `PluginFormatType` 枚举。

除了输入和输出张量之外，不计算所有数据并且需要内存空间的插件可以使用 `getWorkspaceSize` 方法指定额外的内存需求，该方法由构建器调用以确定和预分配暂存空间。

在构建和推理期间，可能会多次配置和执行插件层。在构建时，为了发现最佳配置，层被配置、初始化、执行和终止。为插件选择最佳格式后，再次配置插件，然后在推理应用程序的生命周期内初始化一次并执行多次，最后在引擎销毁时终止。这些步骤由构建器和引擎使用以下插件方法控制：

`configurePlugin` **(配置插件)**

传达输入和输出的数量、所有输入和输出的维度和数据类型、所有输入和输出的广播信息、选择的插件格式和最大批量大小。此时，插件设置其内部状态并为给定配置选择最合适的算法和数据结构。

`initialize` **(初始化)**

此时配置是已知的，并且正在创建推理引擎，因此插件可以设置其内部数据结构并准备执行。

`enqueue` **(排队)**

封装插件的实际算法和内核调用，并提供运行时批处理大小、指向输入、输出和暂存空间的指针，以及用于内核执行的CUDA流。

`terminate` **(终止)**

引擎上下文被销毁，插件持有的所有资源必须被释放。

`clone` **(克隆)**

每次创建包含此插件层的新构建器、网络或引擎时都会调用它。它必须返回一个带有正确参数的新插件对象。

`destroy`**(销毁)**

用于销毁插件对象和/或每次创建新插件对象时分配的其他内存。每当构建器或网络或引擎被破坏时都会调用它。

`set/getPluginNamespace`(设置/获取插件命名空间)

该方法用于设置该插件对象所属的库命名空间（默认可以是""）。来自同一个插件库的所有插件对象都应该具有相同的命名空间。

`IPluginV2Ext`支持可以处理广播输入和输出的插件。此功能需要实现以下方法：

`canBroadcastInputAcrossBatch`

对每个输入调用此方法，其张量在批次中进行语义广播。如果`canBroadcastInputAcrossBatch`返回true （意味着插件可以支持广播），则 TensorRT 不会复制输入张量。插件应该在批处理中共享一个副本。如果它返回false ，则 TensorRT 会复制输入张量，使其看起来像一个非广播张量。

`isOutputBroadcastAcrossBatch`

这为每个输出索引调用。该插件应在给定索引处返回 true 输出，并在整个批次中广播。

`IPluginV2IOExt`
这由构建器在`initialize()`之前调用。它为层提供了基于 I/O PluginTensorDesc和最大批量大小进行算法选择的机会。

注意：基于`IPluginV2`的插件在引擎级别共享，而不是在执行上下文级别共享，因此这些可能被多个线程同时使用的插件需要以线程安全的方式管理它们的资源。创建`ExecutionContext`时会克隆基于`IPluginV2Ext`和派生接口的插件，因此这不是必需的。

9.4.3. IPluginCreator API Description

`IPluginCreator`类中的以下方法用于从插件注册表中查找和创建适当的插件：

`getPluginName`

`IPluginExt::getPluginType`的返回值。

`getPluginVersion`
返回插件版本。对于所有内部 TensorRT 插件，默认为1 。

`getFieldNames`
要成功创建插件，需要了解插件的所有字段参数。此方法返回`PluginFieldCollection`结构，其中填充了`PluginField`条目以反映字段名称和`PluginFieldType`（数据应指向`nullptr`）。

`createPlugin`

此方法用于使用`PluginFieldCollection`参数创建插件。应填充`PluginField`条目的数据字段以指向每个插件字段条目的实际数据。

注意：传递给`createPlugin`函数的数据应该由调用者分配，并在程序被销毁时最终由调用者释放。`createPlugin`函数返回的插件对象的所有权被传递给调用者，并且也必须被销毁。

`deserializePlugin`

此方法由 TensorRT 引擎根据插件名称和版本在内部调用。它应该返回要用于推理的插件对象。在该函数中创建的插件对象在引擎被销毁时被 TensorRT 引擎销毁。

`set/getPluginNamespace`
该方法用于设置此创建者实例所属的命名空间（默认可以是“”）。

9.5. Best Practices For Custom Layers Plugin

调试自定义层问题

必须释放插件中分配的内存以确保没有内存泄漏。如果在`initialize()`函数中获取资源，则需要在`terminate()`函数中释放它们。应该释放所有其他内存分配，最好在插件类析构函数中或在destroy()方法中。使用 C++ API 添加自定义层详细概述了这一点，并提供了一些使用插件时的最佳实践说明。