TensorRT的Python接口解析

本章说明 Python API 的基本用法，假设您从 ONNX 模型开始。 onnx_resnet50.py示例更详细地说明了这个用例。

Python API 可以通过tensorrt模块访问：

```
import tensorrt as trt
```

4.1. The Build Phase

要创建构建器，您需要首先创建一个记录器。 Python 绑定包括一个简单的记录器实现，它将高于特定严重性的所有消息记录到`stdout`。

```
logger = trt.Logger(trt.Logger.WARNING)
```

或者，可以通过从`ILogger`类派生来定义您自己的记录器实现：

```
class MyLogger(trt.ILogger):
    def __init__(self):
       trt.ILogger.__init__(self)

    def log(self, severity, msg):
        pass # Your custom logging implementation here

logger = MyLogger()
```

然后，您可以创建一个构建器：

```
builder = trt.Builder(logger)
```

4.1.1. Creating a Network Definition in Python

创建构建器后，优化模型的第一步是创建网络定义：

```
network = builder.create_network(1 <<
int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
```

为了使用 ONNX 解析器导入模型，需要`EXPLICIT_BATCH`标志。有关详细信息，请参阅显式与隐式批处理部分。

4.1.2. Importing a Model using the ONNX Parser

现在，需要从 ONNX 表示中填充网络定义。您可以创建一个 ONNX 解析器来填充网络，如下所示：

```
parser = trt.OnnxParser(network, logger)
```

然后，读取模型文件并处理任何错误：

```
success = parser.parse_from_file(model_path)
for idx in range(parser.num_errors):
    print(parser.get_error(idx))

if not success:
    pass # Error handling code here
```

4.1.3. Building an Engine

下一步是创建一个构建配置，指定 TensorRT 应该如何优化模型：

```
config = builder.create_builder_config()
```

这个接口有很多属性，你可以设置这些属性来控制 TensorRT 如何优化网络。一个重要的属性是最大工作空间大小。层实现通常需要一个临时工作空间，并且此参数限制了网络中任何层可以使用的最大大小。如果提供的工作空间不足，TensorRT 可能无法找到层的实现：

```
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 20) # 1 MiB
```

指定配置后，可以使用以下命令构建和序列化引擎：

```
serialized_engine = builder.build_serialized_network(network, config)
```

将引擎保存到文件以供将来使用可能很有用。你可以这样做：

```
with open("sample.engine", "wb") as f:
    f.write(serialized_engine)
```

4.2. Deserializing a Plan

要执行推理，您首先需要使用Runtime接口反序列化引擎。与构建器一样，运行时需要记录器的实例。

```
runtime = trt.Runtime(logger)
```

然后，您可以从内存缓冲区反序列化引擎：

```
engine = runtime.deserialize_cuda_engine(serialized_engine)
```

如果您需要首先从文件加载引擎，请运行：

```
with open("sample.engine", "rb") as f:
    serialized_engine = f.read()
```

4.3. Performing Inference

引擎拥有优化的模型，但要执行推理需要额外的中间激活状态。这是通过`IExecutionContext`接口完成的：

```
context = engine.create_execution_context()
```

一个引擎可以有多个执行上下文，允许一组权重用于多个重叠的推理任务。（当前的一个例外是使用动态形状时，每个优化配置文件只能有一个执行上下文。）

要执行推理，您必须为输入和输出传递 TensorRT 缓冲区，TensorRT 要求您在 GPU 指针列表中指定。您可以使用为输入和输出张量提供的名称查询引擎，以在数组中找到正确的位置：

```
input_idx = engine[input_name]
output_idx = engine[output_name]
```

使用这些索引，为每个输入和输出设置 GPU 缓冲区。多个 Python 包允许您在 GPU 上分配内存，包括但不限于 PyTorch、Polygraphy CUDA 包装器和 PyCUDA。

然后，创建一个 GPU 指针列表。例如，对于 PyTorch CUDA 张量，您可以使用`data_ptr()`方法访问 GPU 指针；对于 Polygraphy `DeviceArray`，使用`ptr`属性：

```
buffers = [None] * 2 # Assuming 1 input and 1 output
buffers[input_idx] = input_ptr
buffers[output_idx] = output_ptr
```

填充输入缓冲区后，您可以调用 TensorRT 的`execute_async`方法以使用 CUDA 流异步启动推理。

首先，创建 CUDA 流。如果您已经有 CUDA 流，则可以使用指向现有流的指针。例如，对于 PyTorch CUDA 流，即`torch.cuda.Stream()`，您可以使用`cuda_stream`属性访问指针；对于 Polygraphy CUDA 流，使用`ptr`属性。
接下来，开始推理：

```
context.execute_async_v2(buffers, stream_ptr)
```

通常在内核之前和之后将异步`memcpy()`排入队列以从 GPU 中移动数据（如果数据尚不存在）。

要确定内核（可能还有memcpy()）何时完成，请使用标准 CUDA 同步机制，例如事件或等待流。例如，对于 Polygraphy，使用：

```
stream.synchronize()
```

如果您更喜欢同步推理，请使用`execute_v2`方法而不是`execute_async_v2`。