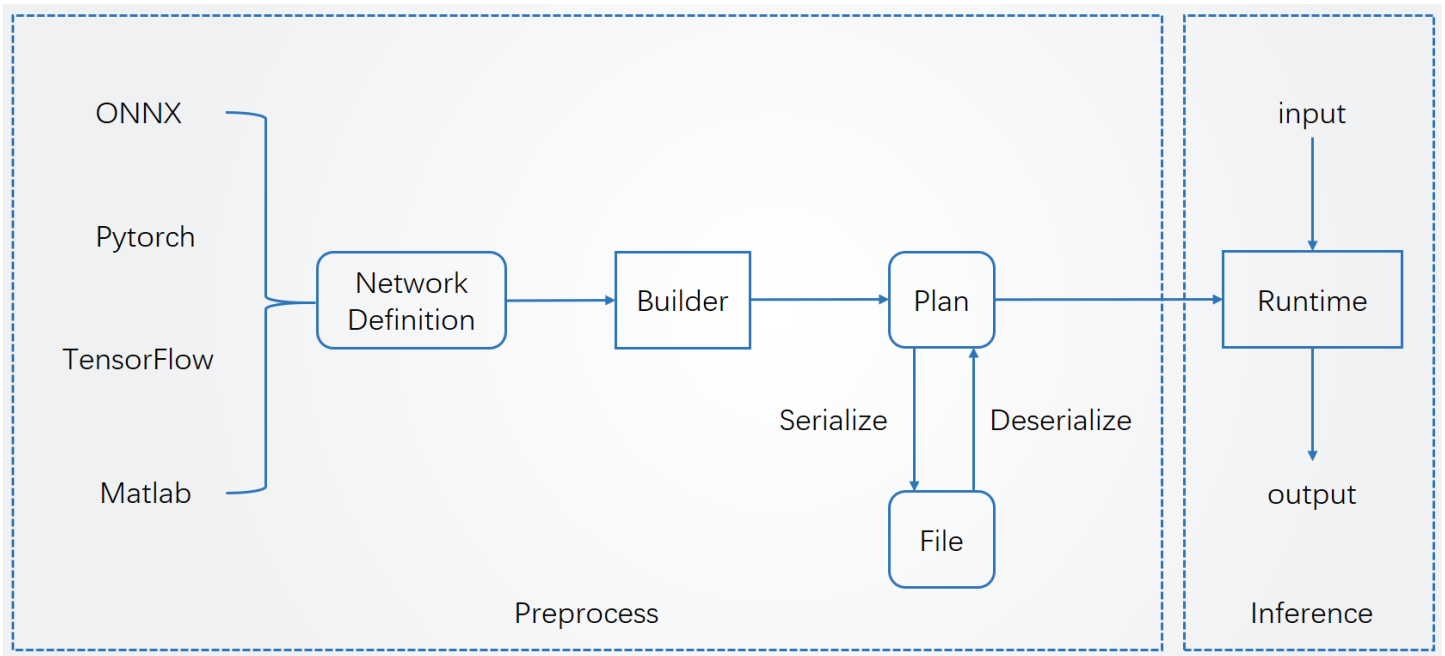
零、基础概念

TensorRT使用流程如下图所示，分为两个阶段：预处理阶段和推理阶段。其部署大致流程如下：1.导出网络定义以及相关权重；2.解析网络定义以及相关权重；3.根据显卡算子构造出最优执行计划；4.将执行计划序列化存储；5.反序列化执行计划；6.进行推理。



可以从步骤3可以得知，tensorrt实际上是和你的硬件绑定的，所以在部署过程中，如果你的硬件（显卡）和软件（驱动、cudatoolkit、cudnn）发生了改变，那么这一步开始就要重新走一遍了。换句话说，不同系统环境下生成Engine 包含硬件有关优化，不能跨硬件平台使用，注意环境统一(硬件环境+ CUDA/cuDNN/TensorRT 环境)，并且不同版本TensorRT 生成的 engine不能相互兼容，同平台同环境多次生成的engine可能不同。

基于TensorRT进行开发的三种工作流程如下图所示：（第5章作业是使用Parser，第6，7章作业是使用TensorRT API开发）

推荐学习资料：

- https://www.bilibili.com/video/BV15Y4y1W73E?spm_id_from=333.337.search-card.all.click
- https://github.com/NVIDIA/trt-samples-for-hackathon-cn/tree/master/cookbook
- https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html (TensorRT 文档)
- https://docs.nvidia.com/deeplearning/tensorrt/api/c_api/ (TensorRT C++ API)
- https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/ (TensorRT Python API)
- https://developer.nvidia.com/nvidia-tensorrt-download （TensorRT 下载）
- https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html (TensorRT 版本支持列表)
-

深蓝学院《CUDA入门与深度神经网络加速》 课程第6、7两章的作业（TensorRT plugin的用法，TensorRT 量化加速）。作业内容是使用TensorRT Python API 手动搭建 BERT 模型。同学们不需要从零 开始搭建，课程组已经搭好了一个模型结构框架，同学们进行相应的填充即可。

模型架构框架：https://github.com/shenlan2017/TensorRT

一、文件信息

1. model2onnx.py 使用pytorch 运行 bert 模型，生成demo 输入输出和onnx模型

2. onnx2trt.py 将onnx模型转成 trt plan 模型，使用onnx-parser转成trt模型，并infer

3. builder.py 输入onnx模型，并进行转换

4. trt_helper.py 对trt的api进行封装，方便调用

5. calibrator.py int8 calibrator 代码

6. 基础款LayerNormPlugin.zip 用于学习的layer_norm_plugin

二、模型信息

2.1 介绍

1. 标准BERT 模型，12 层, hidden_size = 768

2. 不考虑tokensizer部分，输入是ids，输出是score

3. 为了更好的理解，降低作业难度，将mask逻辑去除，只支持batch=1 的输入

BERT模型可以实现多种NLP任务，作业选用了fill-mask任务的模型

```
输入：
The capital of France, [mask], contains the Eiffel Tower.

topk10输出：
The capital of France, paris, contains the Eiffel Tower.
The capital of France, lyon, contains the Eiffel Tower.
The capital of France,, contains the Eiffel Tower.
The capital of France, tolilleulouse, contains the Eiffel Tower.
The capital of France, marseille, contains the Eiffel Tower.
The capital of France, orleans, contains the Eiffel Tower.
The capital of France, strasbourg, contains the Eiffel Tower.
The capital of France, nice, contains the Eiffel Tower.
The capital of France, cannes, contains the Eiffel Tower.
The capital of France, versailles, contains the Eiffel Tower.
```

2.2 输入输出信息

输入

1. input_ids[1, -1]： int 类型，input ids，从BertTokenizer获得

2. token_type_ids[1, -1]：int 类型，全0

3. position_ids[1, -1]：int 类型，[0, 1, ..., len(input_ids) - 1]

输出

1. logit[1, -1, 768]

三、作业内容

（第6章节作业）

3.1 学习使用 trt python api 搭建网络

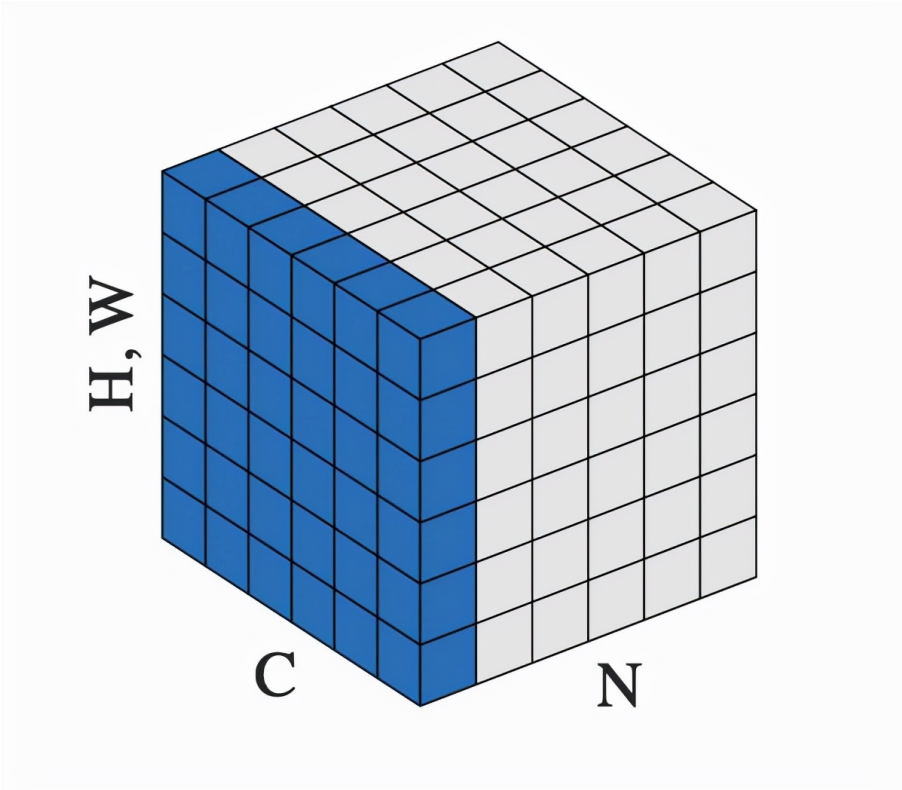
填充trt_helper.py 中的空白函数（addLinear， addSoftmax等）。学习使用api 搭建网络的过程。

思路提示：这部分的主要工作是使用TensorRT Python API实现Bert网络中用到的基本单元操作。那么大家首先要熟悉TensorRT Python API有哪些，这些函数的输入及输出是什么，都要有清晰的认知。详细的可以参看 https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Graph/Network.html 。

另外，builder.py中有关于Bert模型的实现，可以加深对于模型的理解。

addLayerNorm 函数实现

思路提示：LayerNorm是啥?



例如对于输入 x 形状为 (N, C, H, W)， normalized_shape 为 (H, W) 的情况，可以理解为输入 x 为 (N*C, H*W)，在 N*C 个行上，每行有 H*W 个元素，对每行的元素求均值和方差，得到 N*C 个 mean 和 inv_variance，再对输入按如下 LayerNorm 的计算公式计算得到 y。若 elementwise_affine=True ，则有 H*W 个 gamma 和 beta，对每行 H*W 个的元素做变换。

addLinear 函数实现

```
def addLinear(self, x, weight, bias, layer_name=None, precision=None):
```

```
        weight = np.array([weight])
        constant_layer = self.network.add_constant(weight.shape, trt.Weights(weight))
        X_mul = self.network.add_matrix_multiply(x, trt.MatrixOperation.NONE, constant_layer.get_output(0), trt.MatrixOperation.NONE)

        out = X_mul.get_output(0)
        bias = np.array([[bias]])
        bias_layer = self.network.add_constant(bias.shape, trt.Weights(bias))
        trt_layer = self.network.add_elementwise(X_mul.get_output(0), bias_layer.get_output(0), trt.ElementWiseOperation.SUM)
        if layer_name is None:
            layer_name = "trt.addLinear"
        else:
            layer_name = "trt.addLinear." + layer_name

        self.layer_post_process(trt_layer, layer_name, precision)
        return trt_layer.get_output(0)
```

addSoftmax 函数实现

```
def addSoftmax(self, x: trt.ITensor, dim: int = -1, layer_name=None, precision=None) -> trt.ITensor:
        trt_layer = self.network.add_softmax(x)

        input_len = len(x.shape)
        if dim is -1:
            dim = input_len
        trt_layer.axes = int(math.pow(2, input_len-1))

        layer_name_prefix = "nn.Softmax[dim=" + str(dim) + "]"
        if layer_name is None:
            layer_name = layer_name_prefix
        else:
            layer_name = layer_name_prefix + "." + layer_name

        self.layer_post_process(trt_layer, layer_name, precision)

        return trt_layer.get_output(0)
```

addScale 函数实现

```python
def addScale(self, x: trt.ITensor, scale: float, layer_name: str = None, precision: trt.DataType = None) -> trt.ITensor:
        """scale"""
        input_len = len(x.shape)
        if input_len < 3:
            raise RuntimeError("input_len < 3 not support now! ")

        if layer_name is None:
            layer_name = "Scale"

        # The input dimension must be greater than or equal to 4
        if input_len is 3:
            trt_layer = self.network.add_shuffle(x)
            trt_layer.reshape_dims = (0, 0, 0, 1)
            self.layer_post_process(trt_layer, layer_name+".3dto4d", precision)
            x = trt_layer.get_output(0)

        np_scale = trt.Weights(np.array([scale], dtype=np.float32))
        trt_layer = self.network.add_scale(x, mode=trt.ScaleMode.UNIFORM,
                                           shift=None, scale=np_scale, power=None)
        self.layer_post_process(trt_layer, layer_name, precision)
        x = trt_layer.get_output(0)

        if input_len is 3:
            trt_layer = self.network.add_shuffle(x)
            trt_layer.reshape_dims = (0, 0, 0)
            self.layer_post_process(trt_layer, layer_name+".4dto3d", precision)
            x = trt_layer.get_output(0)

        return x
```

addMatMul 函数实现

```python
def addMatMul(self, a: trt.ITensor, b: trt.ITensor, layer_name: Optional[str] = None) -> trt.ITensor:
```

```
        trt_layer = self.network.add_matrix_multiply(a, trt.MatrixOperation.NONE, b, trt.MatrixOperation.NONE)

        if layer_name is None:
            layer_name = "torch.matmul"
        else:
            layer_name = "torch.matmul." + layer_name

        self.layer_post_process(trt_layer, layer_name, None)

        return trt_layer.get_output(0)
```

3.2 编写plugin

trt不支持layer_norm算子，编写layer_norm plugin，并将算子添加到网络中，进行验证。

1. 及格：将 “基础款LayerNormPlugin.zip”中实现的基础版 layer_norm算子 插入到 trt_helper.py addLayerNorm函数中。

思路提示：首先将老师提供的代码进行编译，然后再将算子插入到addLayerNorm函数中。

```
def getLayerNormPlugin():
    for c in trt.get_plugin_registry().plugin_creator_list:
        if c.name == 'LayerNorm':
            return c.create_plugin(c.name, trt.PluginFieldCollection([]))
    return None

def addLayerNorm(self, x, gamma, beta, layer_name=None, precision=None):
        # TODO: create your layer norm plugin
        inputTensorList = []
        inputTensorList.append(x)
        pluginLayer = self.network.add_plugin_v2(inputTensorList, getLayerNormPlugin())

        self.network.mark_output(pluginLayer.get_output(0))
        if layer_name is None:
            layer_name = "trt.LayerNorm"
        else:
            layer_name = "trt.LayerNorm." + layer_name

        self.layer_post_process(pluginLayer, layer_name, precision)
```

```
20          return pluginLayer.get_output(0)
```

2. 优秀：将整个layer_norm算子实现到一个kernel中，并插入到 trt_helper.py addLayerNorm函数中。可以使用testLayerNormPlugin.py对合并后的plugin进行单元测试验证。
3. 进阶：在2的基础上进一步优化，线索见 https://www.bilibili.com/video/BV1i3411G7vN

3.3 观察GELU算子的优化过程

1. GELU算子使用一堆基础算子堆叠实现的（详细见trt_helper.py addGELU函数），直观上感觉很分散，计算量比较大。
2. 但在实际build过程中，这些算子会被合并成一个算子。build 过程中需要设置log为trt.Logger.VERBOSE，观察build过程。
3. 体会trt在转换过程中的加速优化操作。

思路提示：理解addGELU函数的操作，然后对应于Build过程中的相关操作。

四、作业内容（第7章节作业）

需要根据前面的提示，完成一些初步的工作。

4.1 进行 fp16 加速并测试速度

及格标准：设置build_config，对模型进行fp16优化；
思路提示：认真阅读老师提供的代码，修改builder.sh中的输入参数。

优秀标准：编写fp16 版本的layer_norm算子，使模型最后运行fp16版本的layer_norm算子。
思路提示：layer_norm 核函数的实现时，使用函数模板来处理，通过将类型作为参数传递给模板，可使编译器生成该类型的函数。

4.2 进行 int8 加速并测试速度

完善calibrator.py内的todo函数，使用calibrator_data.txt 校准集，对模型进行int8量化加速。

思路提示：将FP32降为INT8的过程相当于信息再编码（re-encoding information ），就是原来使用32bit来表示一个tensor，现在使用8bit来表示一个tensor，还要求精度不能下降太多。

这部分整体还是基本的操作，主要完成TODO部分工作即可。
首先是读入calibrator_data.txt文件，转化为模型的输入数据。

```
def text2inputs(tokenizer, text):
    encoded_input = tokenizer.encode_plus(text, return_tensors = "pt")

    input_ids = encoded_input['input_ids'].int().detach().numpy()
    token_type_ids = encoded_input['token_type_ids'].int().detach().numpy()
    # position_ids = torch.arange(0, encoded_input['input_ids'].shape[1]).int().view(1, -1).numpy()
    seq_len = encoded_input['input_ids'].shape[1]
    position_ids = np.arange(seq_len, dtype = np.int32).reshape(1, -1)
    input_list = [input_ids, token_type_ids, position_ids]

    return input_list

# TODO: your code, read inputs
with open(data_txt, "r") as f:
            lines = f.readlines()
            for i in range(0, num_inputs):
                inputs = text2inputs(tokenizer, lines[i])
                self.input_ids_list.append(inputs[0])
                self.token_type_ids_list.append(inputs[1])
                self.position_ids_list.append(inputs[2])
                if i % 10 == 0:
                    print("text2inputs:" + lines[i])
```

然后是get_batch()中的代码补全，这里贴出参考实现代码，如下：

```
# TODO your code, copy input from cpu to gpu
input_ids = self.input_ids_list[self.current_index]
token_type_ids = self.token_type_ids_list[self.current_index]
position_ids = self.position_ids_list[self.current_index]

seq_len = input_ids.shape[1]
if seq_len > self.max_seq_length:
    print(seq_len)
    print(input_ids.shape)
    input_ids = input_ids[:, :self.max_seq_length]
```

```
    token_type_ids = token_type_ids[:, :self.max_seq_length]
    position_ids = position_ids[:, :self.max_seq_length]
    print(input_ids.shape)

cuda.memcpy_htod(self.device_inputs[0], input_ids.ravel())
cuda.memcpy_htod(self.device_inputs[1], token_type_ids.ravel())
cuda.memcpy_htod(self.device_inputs[2], position_ids.ravel())

self.current_index += self.batch_size
```

参考资料：

- https://github.com/NVIDIA/TensorRT （TensorRT GitHub，特别是demo 和samples部分）
- https://docs.nvidia.com/deeplearning/tensorrt/index.html （TensorRT 官方文档）