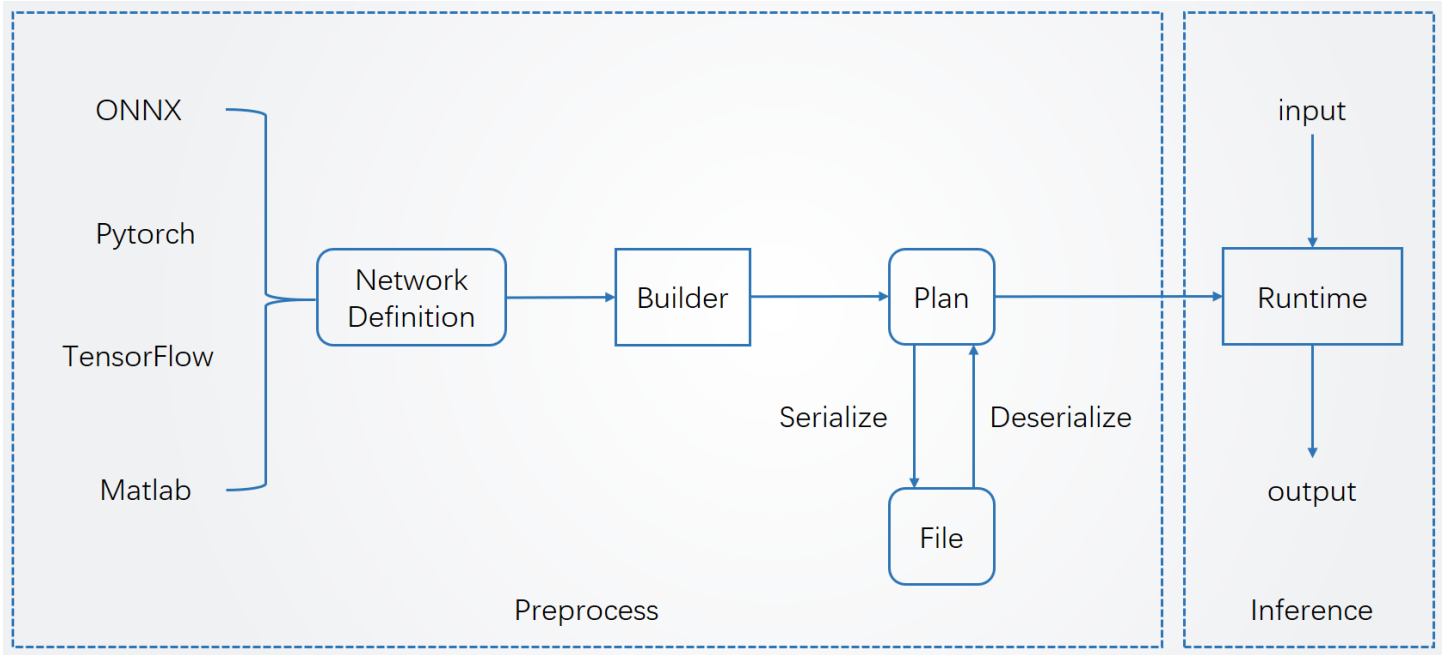
零、基础概念

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)

一、背景介绍

深蓝学院《CUDA入门与深度神经网络加速》 课程，TensorRT 部分的作业。

内容是使用TensorRT Python API 手动搭建图像识别Vision Transformer(ViT)模型。学员不需要从零开始搭建，这里已经搭好了一个模型结构框架，进行相应的填充即可。

ViT 模型知识给大家推荐一些博客文章，请大家自行学习：

- Vision Transformer 超详细解读系列文章 https://zhuanlan.zhihu.com/p/340149804
- 详解 Vision Transformer (ViT) https://blog.csdn.net/qq_39478403/article/details/118704747

> 注意： 整个作业只支持batch=1

二、模型数据介绍和下载

2.1 模型下载

github上有多个ViT项目，这里选择了一个最适合学习的项目（已经下载了，见压缩包ViT-pytorchmain.zip）。

```
https://github.com/jeonsworld/ViT-pytorch
model_type: ViT-B_16
dataset: cifar10-100_500
input: [batch, 3, 224, 224]
output: [batch, 10]
```

但这个项目有个比较大的缺点，只提供了pre-train和 fine-tuned 的npy格式的模型，识别率为0，无法直接使用。有两个改进方案：
方案1，按照该项目readme中介绍的使用方法，训练出来一个模型，并转成onnx模型。
方案2，课程提供了一个只训练了100 step的模型，一个checkpoint模型和一个转好的onnx模型。
文件下载地址：https://pan.baidu.com/s/1StW6Z4yy8uTklR52X9K8Cg?pwd=m3w7，提取码：m3w7。

2.2 数据下载

使用 CIFAR10-100_500 数据集。
1. github上的项目，训练时为了提高读取文件的效率，采用的是将多张图片合并成一个文件的形式。https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
2. 咱们作业使用的更直观的 jpg 格式的数据，共十类，每类1000张图片。https://github.com/YoongiKim/CIFAR-10-images

2.3 验证base准确性

1. 使用 onnx 模型和 jpg 格式的数据 验证模型准确性。
2. 方案2提供的模型，识别准确率为89.72% （很低，因为只训练了100 step）
3. 验证命令

```
python valid.py -x ViT-B_16.onnx -d CIFAR-10-images-master/test/
```

2.4 测速

1. 测试 onnx cpu 速度 （选做）
2. 测试 onnx gpu 速度 （选做）
3. 测试 使用trt onnxparser 直接转出来的trt模型速度
4. 测试 手动转换出来的 不同精度 trt 模型的速度

trt 模型测速方式，建议使用trtexec，选择 Host Latency or GPU Compute 的 mean 时间即可：

```
./trtexec --loadEngine=model.plan --shapes=input:1x1x224x224 --plugins=LayerNorm.so
```

三、目录文件介绍

```
ViT-pytorch # https://github.com/jeonsworld/ViT-pytorch
model2onnx.py # 将 npy 模型 转成 onnx 模型的代码，注意重要的参数do_constant_folding=False

# layernorm 相关
LayerNormPlugin
test_nn_layer_norm.py
test_nn_layer_norm.sh

# 验证onnx和trt模型准确率的脚本
valid.sh
valid.py

# onnx 模型转成 trt 格式的相关
# builder.py 中的结构和 ViT-pytorch/models/modeling.py 差不多。
builder.sh
builder.py
trt_helper.py
calibrator.py
ViT-B_16.onnx

# 这个脚本很重要，支持load checkpoint 格式模型，并跑一张图片
# 当转出来的 trt 模型结果对不上时，可以使用这个脚本进行调试。
# 使用方法自己琢磨吧。
test.py
```

四、作业内容

0. 使用trt onnx-parser 将 onnx 模型转成 trt 格式，并测速。

```
#使用 onnx-parser将onnx模型转换成trt plan格式
#sh builder.sh
#验证
python valid.py -p model.plan -d CIFAR-10-images-master/test/
```

思路提示:

首先，获取onnx模型，有两种方式：

- 使用老师提供的onnx模型
- 或者执行model2onnx.py自己生成出onnx模型

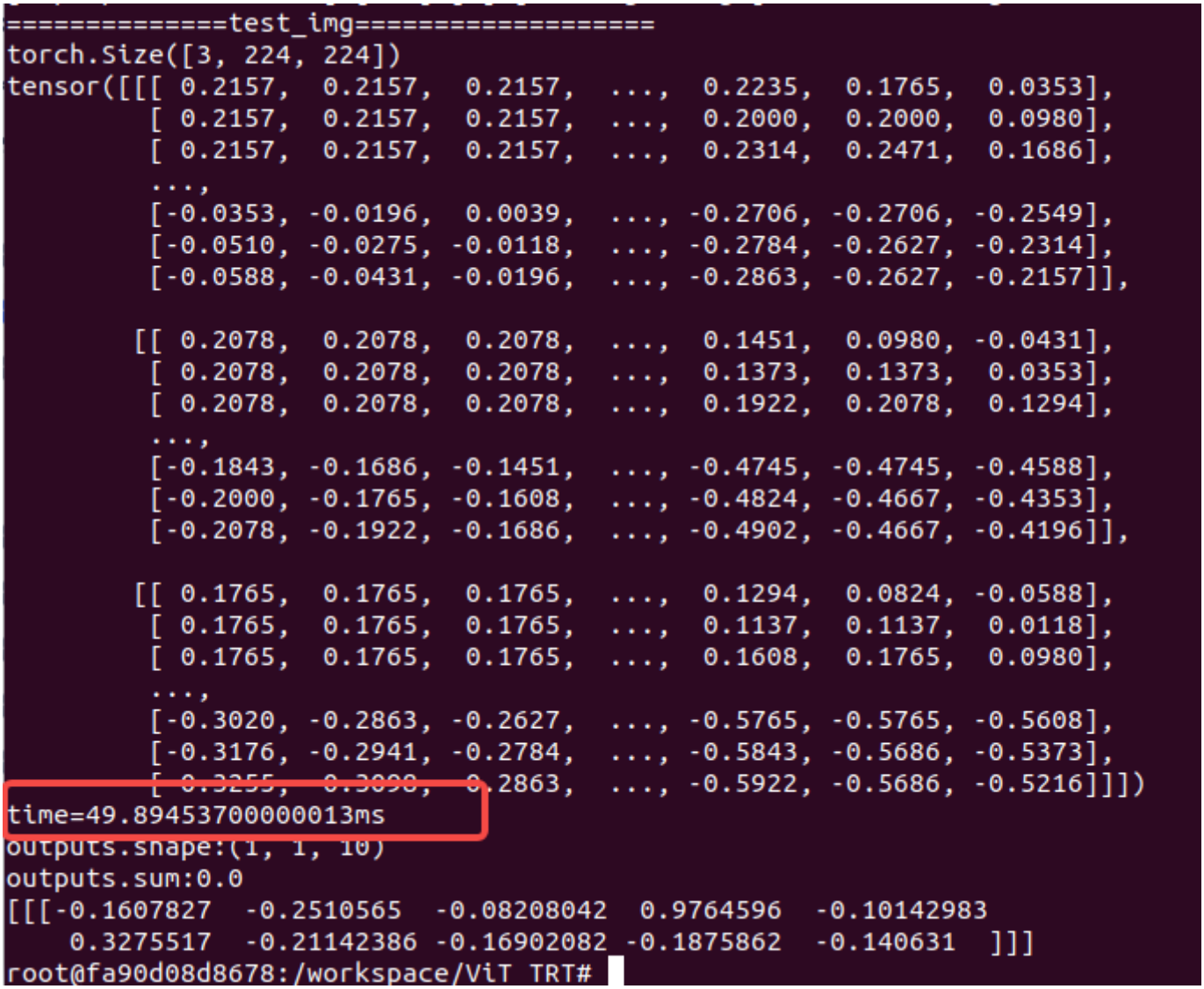
然后，将 onnx 模型转成 trt 格式（可以参考第5章作业），并测速。

执行如下命令，实现转换。如需测试，则徐指定一张图片的路径。

```
python builder.py -x ./models/ViT-B_16.onnx -o ./plans/model.plan

python builder.py -x ./models/ViT-B_16.onnx -o ./plans/model.plan --img_path ViT-pytorch/00816.jpeg
```

```
[Constant]_output[Float(768)] -> (Unnamed Layer* 572) [PluginV2DynamicExt]_output_0[Float(1,197,768)]
Layer(Reformat): 575_nn.Linear_pre_reshape_copy_input, Tactic: 0x00000000000003e8, (Unnamed Layer* 572) [PluginV2DynamicExt]_output_0[Float(1,1,768)] -> 575_
,1,768)]
Layer(NoOp): Reformatting CopyNode for Input Tensor 0 to 575_nn.Linear_pre_reshape, Tactic: 0x0000000000000000, 575_nn.Linear_pre_reshape_copy_input[Float(1,
 575_nn.Linear_pre_reshape[Float(1,1,768)]
Layer(NoOp): 575_nn.Linear_pre_reshape, Tactic: 0x0000000000000000, Reformatted Input Tensor 0 to 575_nn.Linear_pre_reshape[Float(1,1,768)] -> (Unnamed Layer
1)]
Layer(CublasConvolution): 576_nn.Linear, Tactic: 0x0000000000000001, (Unnamed Layer* 574) [Shuffle]_output[Float(1,1,768,1,1)] -> (Unnamed Layer* 575) [Fully
Layer(NoOp): 577_nn.Linear_after_reshape, Tactic: 0x0000000000000000, (Unnamed Layer* 575) [Fully Connected]_output[Float(1,1,10,1,1)] -> (Unnamed Layer* 576
[09/15/2022-18:39:41] [TRT] [I] [MemUsageChange] TensorRT-managed allocation in building engine: CPU +0, GPU +0, now: CPU 0, GPU 0 (MiB)
[09/15/2022-18:39:41] [TRT] [I] build engine in 12.403 Sec
[09/15/2022-18:39:41] [TRT] [V] Serializing Engine...
[09/15/2022-18:39:41] [TRT] [W] The getMaxBatchSize() function should not be used with an engine built from a network created with NetworkDefinitionCreationF
 will always return 1.
[09/15/2022-18:39:41] [TRT] [W] The getMaxBatchSize() function should not be used with an engine built from a network created with NetworkDefinitionCreationF
 will always return 1.
[09/15/2022-18:39:41] [TRT] [I] Saving Engine to ./plans/model.plan
[09/15/2022-18:39:41] [TRT] [I] Done.
root@fa90d08d8678:/workspace/ViT_TRT#
```



1. 学习 使用trt python api 搭建网络

填充trt_helper.py 中的空白函数，包括Linear，LayerNorm，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中有关于模型的实现，可以加深对于模型的理解。

```
def addLinear(self, x, weight, bias, layer_name=None, precision=None):
        input_len = len(x.shape)
        if input_len < 3:
            raise RuntimeError("addLinear x.shape.size must >= 3")

```

```
        if layer_name is None:
            layer_name = "nn.Linear"

        # calc pre_reshape_dims and after_reshape_dims
        pre_reshape_dims = trt.Dims()
        after_reshape_dims = trt.Dims()
        if input_len == 3:
            pre_reshape_dims = (0, 0, 0, 1, 1)
            after_reshape_dims = (0, 0, 0)
        elif input_len == 4:
            pre_reshape_dims = (0, 0, 0, 0, 1, 1)
            after_reshape_dims = (0, 0, 0, 0)
        elif input_len == 5:
            pre_reshape_dims = (0, 0, 0, 0, 0, 1, 1)
            after_reshape_dims = (0, 0, 0, 0, 0)
        else:
            raise RuntimeError("addLinear x.shape.size >5 not support!")

        # add pre_reshape layer
        trt_layer = self.network.add_shuffle(x)
        trt_layer.reshape_dims = pre_reshape_dims

        self.layer_post_process(trt_layer, layer_name+"_pre_reshape", precision)

        x = trt_layer.get_output(0)

        # add Linear layer
        out_features = weight.shape[0]
        weight = trt.Weights(weight)
        if bias is not None:
            bias = trt.Weights(bias)

        trt_layer = self.network.add_fully_connected(x, out_features, weight, bias)
        self.layer_post_process(trt_layer, layer_name, precision)
        x = trt_layer.get_output(0)

        # add after_reshape layer
        trt_layer = self.network.add_shuffle(x)
        trt_layer.reshape_dims = after_reshape_dims
```

```
        self.layer_post_process(trt_layer, layer_name+"_after_reshape", precision)
        x = trt_layer.get_output(0)

        return x
```

```
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)

        x = trt_layer.get_output(0)
        return x
```

```
def addAdd(self, a, b, layer_name=None, precision=None):
        trt_layer = self.network.add_elementwise(a, b, trt.ElementWiseOperation.SUM)
        if layer_name is None:
            layer_name = "elementwise.sum"
        else:
            layer_name = "elementwise.sum." + layer_name

        self.layer_post_process(trt_layer, layer_name, precision)

        x = trt_layer.get_output(0)
        return x
```

```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
```

```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)

        x = trt_layer.get_output(0)
        return x
```

2. 编写layernorm 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.pyaddLayerNorm函数中。

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?spm_id_from=333.999.0.03.

思路提示:

首先，要了解LayerNorm是啥，什么原理，是怎么操作实现的。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。

然后，要编译基础版LayerNormPlugin编译，注意修改Makefile中 CUDA_PATH，TRT_PATH的值，对应于自己的环境进行配置。执行编译后，得到如下的情况

然后，再将算子插入到addLayerNorm函数中。

```
def addLayerNorm(self, x, gamma, beta, layer_name=None, precision=None):
        plg_creator = self.plugin_registry.get_plugin_creator("LayerNorm", "1", "")
        if not plg_creator:
            raise RuntimeError("Could not find LayerNorm")
```

```
5
6           # pfc = trt.PluginFieldCollection([data_type, dim, eps, gamma_w, beta_w])
7           pfc = trt.PluginFieldCollection([])
8           plugin = plg_creator.create_plugin("LayerNorm", pfc)
9           if not plugin:
10              raise RuntimeError("Could not create_plugin LayerNormPluginDynamic")
11
12          gamma = self.network.add_constant(gamma.shape, gamma).get_output(0)
13          beta = self.network.add_constant(beta.shape, beta).get_output(0)
14
15          trt_layer = self.network.add_plugin_v2([x, gamma, beta], plugin)
16
17          if layer_name is None:
18              layer_name = "nn.LayerNorm"
19
20          self.layer_post_process(trt_layer, layer_name, precision)
21
22          return trt_layer.get_output(0)
```

3. 观察GELU算子的优化过程

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。
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。

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。

4. 学习 builder.py

重点是梳理build_embeddings_layer函数逻辑，体会pytorch api 和 trt api 的差异。

5. 进行 fp16 加速，观察模型大小，准确率和速度

需要注意plugin 是否支持 fp16？是否设置了fp16?

1. 及格：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。

2. 优秀：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。

6. 进行 int8 加速，观察模型大小，准确率和速度

完善calibrator.py内的todo函数
1. int8 出来的模型，准确率可能会掉很多，为什么
2. int8 和 fp16 可以都 enable，观察 模型大小，准确率和速度

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。

```
classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
def read_all_test_imgs(args, num_inputs_per_class):
    imgs = []
    labels = []
    label_idx = 0
    class_count = 0
    for c in classes:
        path = args.calib_path+ "/" + c
        for i in os.listdir(path):
            #  print(path + "/" + i)
            #  assert 0
            img = image_preprocess(args, path + "/" + i).numpy()
            imgs.append(img)
            labels.append(label_idx)
            class_count = class_count + 1
            if class_count >= num_inputs_per_class:
                break
        label_idx = label_idx + 1
        print(f"read {c} done ..., img num = {class_count}")
        class_count = 0
```

```
    return imgs, labels

```

```
class ViTCalibrator(trt.IInt8LegacyCalibrator):
    def __init__(self, args, cache_file, batch_size, num_inputs):
        # Whenever you specify a custom constructor for a TensorRT class,
        # you MUST call the constructor of the parent explicitly.
        trt.IInt8LegacyCalibrator.__init__(self)

        self.imgs, _ = read_all_test_imgs(args, num_inputs)

        self.cache_file = cache_file

        self.batch_size = batch_size
        self.current_index = 0
        if num_inputs > len(self.imgs):
            self.num_inputs = len(self.imgs)
        else:
            self.num_inputs = num_inputs

        # Allocate enough memory for a whole batch.
        self.device_input = cuda.mem_alloc(self.batch_size * self.imgs[0].nbytes)

```

```
def get_batch(self, names):
        if self.current_index + self.batch_size > self.num_inputs:
            print("Calibrating index {:} batch size {:} exceed max input limit {:} sentences".format(self.current_index, self.batch_size, self.num_inputs))
            return None

        current_batch = int(self.current_index / self.batch_size)
        if current_batch % 10 == 0:
            print("Calibrating batch {:}, containing {:} imgs".format(current_batch, self.batch_size))

        #  import pdb
```

```
        #  pdb.set_trace()
        cuda.memcpy_htod(self.device_input, self.imgs[self.current_index].ravel())

        self.current_index += self.batch_size
        return [self.device_input]
```

```
if __name__ == '__main__':
    data_txt = "calibrator_data.txt"
    bert_path = "bert-base-uncased"
    cache_file = "bert_calibrator.cache"
    batch_size = 1
    max_seq_length = 200
    num_inputs = 100
    cal = BertCalibrator(data_txt, bert_path, cache_file, batch_size, max_seq_length, num_inputs)

    cal.get_batch("input")
    cal.get_batch("input")
    cal.get_batch("input")
```

7. 支持batch_size > 1(选做)

可以按照以下提示进行修改

```
1. 修改 builder.py 中的 profile 和valid.py
2. builder.py build_embeddings_layer 函数中，cls_token 的batch 维度只有1
3. builder.py build_vision_transformer_layer函数中，Slice batch维度
4. calibrator.py 中，需要进行 batch 拼接。
```