Lab 6：使用 NaiveTPU 实现神经网络推理

1 实验目标

1) 使用 NaiveTPU 实现 MLP、LeNet 神经网络推理

2 实验环境

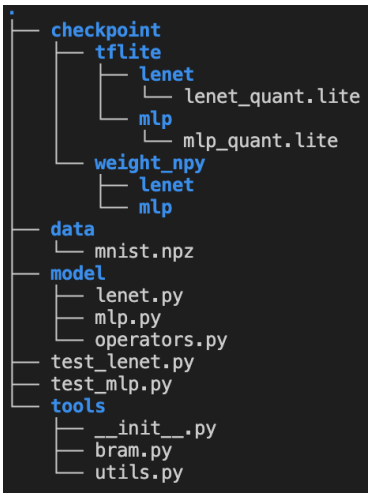
1) Vivado 2019.2 / Vitis 2019.2
2) ZYNQ 7020 开发板及其配件
3) ZYNQ 上的 Linux 系统
4) ZYNQ 中的 Python 以及 Numpy 库
5) 本实验过程中使用到的数据文件可在北航盘下载：
   https://bhpan.buaa.edu.cn/link/AA6CBFEEE611D54668AD5E0EB55521DA83
   Name: Lab-6：使用 NaiveTPU 实现神经网络推理（实验材料）.rar
   Expires: 2025-03-01 16:10
   Pickup Code: 3XlG
6) 注意：本实验指导书中给出的步骤仅为示意步骤作为参考，每人遇到的情况可能有差异，如果遇到问题可根据实际情况进行探索，或向助教寻求帮助。

3 实验要求

1) 按照 5.1 中的实验步骤完成实验，并得到正确的测试结果。
2) 回答 5.2 中提出的问题。（必做）
3) 回答 5.3 中的附加题，会根据回答结果酌情考虑加分。（选做）
4) 撰写实验报告。实验报告命名：学号+姓名+实验六。（实验报告撰写细节可参考“实验报告撰写格式”）
5) 将实验报告与工程文件压缩包，按时交至 SPOC 课程作业处。

4 实验说明

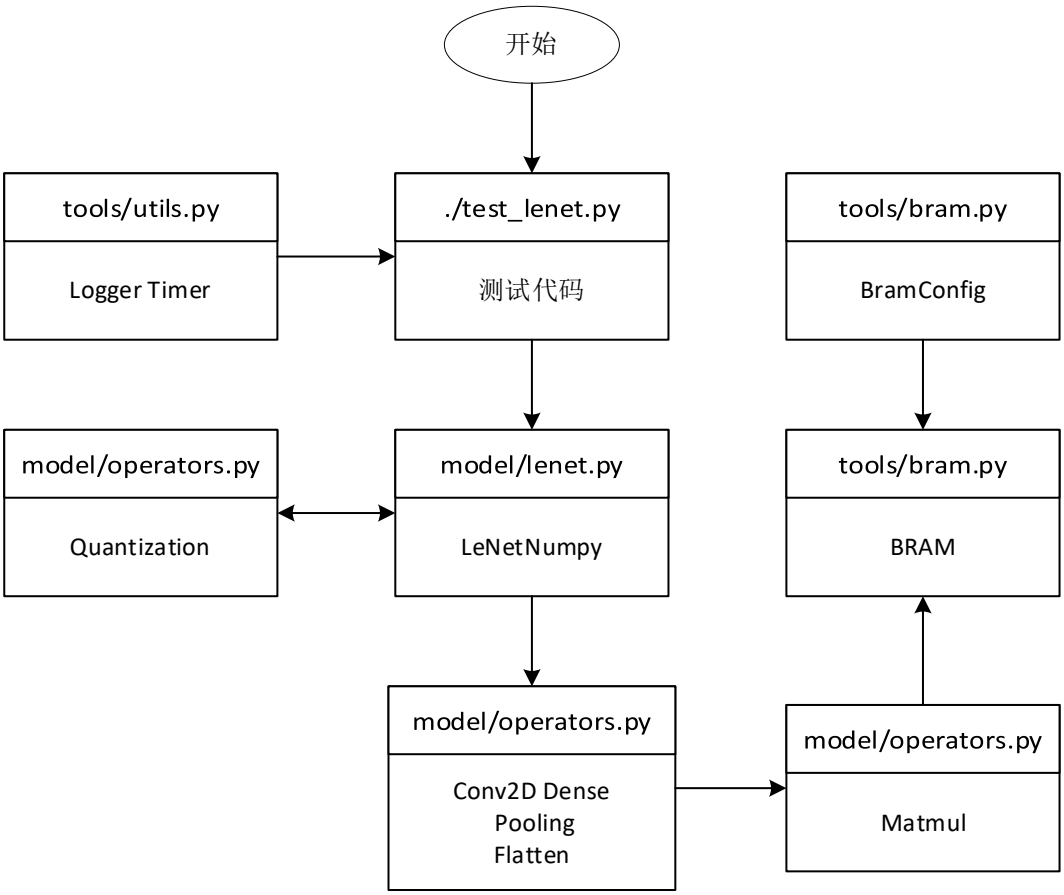
4.1 代码文件目录结构说明



- ./：根目录
  - test_mlp.py、test_lenet.py：测试代码入口文件
- checkpoint：存储模型及权重等参数
  - tflite：存储 tensorflow lite 量化后的模型，可以使用 Netron 打开查看模型结构、权重、量化参数等信息
  - weight_npy：存储模型权重信息
- data：存储数据集
- model：模型类实现、神经网络量化推理算子实现
  - mlp.py、lenet.py：模型类实现
  - operators.py：神经网络量化推理算子实现、Matmul 实现、量化参数
- tools：存储工具函数和类
  - bram.py：BRAM 配置、读写
  - utils.py：工具函数/类（Logger、FLAG、Timer、load_mnist）

4.2 代码介绍

代码调用堆栈如下图所示：



代码运行的测试入口为根目录下的test_mlp.py或者test_lenet.py，上图中以test_lenet.py为例，但是 mlp 网络同理。接下来对网络测试主要流程进行说明。

4.2.1 test_lenet.py

该代码为测试神经网络的主要入口代码，首先加载数据集：

```
###################### 准备数据集 ######################
(x_train, y_train), (x_test, y_test) = load_mnist('data/mnist.npz')
x_test = x_test / 255.
```

创建网络、时间统计器的实例，网络实例用于执行推理的时候调用，时间统计器用于统计神经网络推理时间。

```
###################### 创建实例 ######################
net = mlp.MLPNumpy('Matmul')
timer = Timer()
```

执行神经网络推理并计算推理延时和准确率。

```
#################### 测试模型 ####################
correct = 0
samples_num = 20 #len(x_test)
for i in range(samples_num):
    logger.info("{}/{}".format(i, samples_num))                    ①

    image = x_test[i][np.newaxis, :]
    label = y_test[i]    # 这里的label就是一个数字，如7，表示这张图片是7

    # 推理并记录时间
    timer.start()
    prediction = net(image)                                        ②
    timer.end()
    logger.info("Inference time: {}".format(timer.current_time()))
    logger.info("Average time: {}".format(timer.avg_time()))

    # 判断推理正确性
    if np.argmax(prediction, axis=1)[0] == label:                  ③
        correct += 1
logger.info('Accuracy: {}%'.format(correct / samples_num * 100))
logger.info("Average time: {}".format(timer.avg_time()))
```

① 从测试集中取出输入图片和标签
② 神经网络执行推理
③ 判断此次推理的正确性

4.2.2 model/lenet.py

该文件中 LeNetNumpy 类用于实现 LeNet 量化推理函数，其中__init__方法执行一些初始化操作，forward 方法执行网络前向传播操作。

__init__方法中拥有一个输入参数，matmul，该参数为字符串，用于决定矩阵乘法的方式，传入 np 使用 np.matmul 方式；传入 Matmul 则使用同学们 Lab4、Lab5 中自己实现的 Matmul 接口。（请注意，需要同学们用自己的 Matmul 类实现替换 model/operators.py 文件中的 Matmul 类）

```
def __init__(self, matmul: str):
    '''
        输入:
            matmul: np, Matmul
    '''
```

根据输入字符串选择矩阵乘法方式：

```
if matmul == 'np':
    self.matmul = np.matmul  # 矩阵乘法方式定义
elif matmul == 'Matmul':
    self.matmul = Matmul()  # 矩阵乘法方式定义
else:
    raise Exception('matmul type does not exist')
```

读取神经网络参数，包括各层的 weight 和 bias。

```
self.weight_dir = os.path.join(os.getcwd(), 'checkpoint/weight_npy/lenet')
self.w1 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_conv2d_Conv2D.npy'))
self.b1 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_conv2d_BiasAdd;le_net_sequential_conv2d_Conv2D;conv2d_bias.npy'))
self.w2 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_conv2d_1_Conv2D.npy'))
self.b2 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_conv2d_1_BiasAdd;le_net_sequential_conv2d_1_Conv2D;conv2d_1_bias.npy'))
self.w3 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_dense_MatMul.npy'))
self.b3 = np.load(os.path.join(self.weight_dir, 'dense_bias.npy'))
self.w4 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_dense_1_MatMul.npy'))
self.b4 = np.load(os.path.join(self.weight_dir, 'dense_1_bias.npy'))
self.w5 = np.load(os.path.join(self.weight_dir, 'le_net_sequential_dense_2_MatMul.npy'))
self.b5 = np.load(os.path.join(self.weight_dir, 'dense_2_bias.npy'))
```

量化参数定义，在本课程中，只有全连接层（Dense）和卷积层（Conv2D）需要传入量化参数（*想了解量化推理过程的同学可以查看 PPT*）：

```
# 量化参数定义
self.conv1_quantization_params = Quantization(
    scale={
        'input': 0.003921568859368563,
        'weight': [0.0024505917, 0.001674911, 0.0031223092, 0.004236928, 0.003395076, 0.004709337],
        'output': 0.00864633172750473
    },
    zero_point={'input': -128, 'weight': 0, 'output': -128}
)
```

定义网络所需要的算子，此处以 Pooling 为例：（请注意，此处需要同学们仿照 Pooling 算子的定义补全网络中使用到的其它算子的定义）

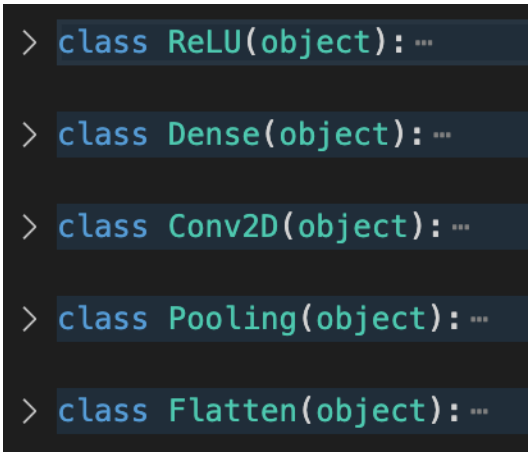
```
# 网络层定义
# 例:
self.maxpooling = Pooling(ksize=(2,2), method='max')
```

forward()函数中使用定义好的算子执行网络前向传播过程，即网络推理过程。（此处需要同学们根据网络结构使用上一步定义的算子完成网络前向传播过程）

```
def forward(self, x):
    # 前向传播
    # 例:
    output = self.maxpooling(x)
    return output
```

4.2.3 model/operators.py

该文件中包含神经网络量化推理算子的具体实现、Matmul 矩阵乘法类、Quantization 量化参数收集处理类。其中，量化推理算子包含：



其中，需要注意，ReLU 算子不需要使用，ReLU 过程已经蕴含在了各个量化推理过程中。

Flatten()算子使用方法：

```
flatten = Flatten()
output = flatten(input)
```

Pooling 算子使用方法：

```
maxpooling = Pooling(ksize=(2,2), method='max', pad=False)
output = maxpooling(input)
```

其中，各参数含义如下：

- ksize：滤波器大小
- method：max 或者 mean，即选择最大池化或者平均池化
- pad：是否进行 padding 操作，即使得输入和输出大小保持一致

Dense 算子使用方法：

```
dense = Dense(
    weight,
    bias,
    quantization_parameters,
    matmul=matmul
)
output = dense(input)
```

其中，各参数含义如下：

- weight：权重
- bias：偏移值
- quantization_parameters：量化参数，Quantization 类型
- matmul：矩阵乘法接口，np.matmul 或者 Matmul()

Conv2D 算子使用方法：

```
dense = Conv2D(
    weight,
    bias,
    quantization_parameters,
    pad='VALID',
    stride=(1,1),
    matmul=matmul
)
output = dense(input)
```

其中，各参数含义如下：

- weight：权重
- bias：偏移值
- quantization_parameters：量化参数，Quantization 类型
- pad：VALID 模式或者 SAME 模式

4.2.4 tools/bram.py

bram.py 中实现了 BramConfig 以及 BRAM 读写类，该部分中在 Lab4 中已经介绍过了，此处不再复述。此处说明一下代码变化的部分。

1) BRAM 类中 read 接口变化：

```
def read(self, len, block_name, offset='default', dtype=np.uint8) -> np.ndarray:
    '''按字节依次从低字节读取

        Args:
            len: 读取数据长度，单位字节
            block_name: BramConfig中配置的block_info的key值
            offset: 支持两种输入模式
                    1. str: BramConfig中配置的offset字典key值
                    2. int: 在block上的偏移量
            dtype: 要求数据按相应的格式输出，
                    np.int8, np.int16, np.int32, np.int64,
                    np.uint8, np.uint16, np.uint32, np.uint64

        Return:
            np.adarray
    '''
```

read 接口中 offset 支持两种输入模式，除了 BramConfig 中配置的 offset 的 key 值外，可以传入类型为 int 的值，表示将名称为 block_name 的 BRAM 块内的偏移量作为起始地址开始读数据。（仅作为接口扩展，非必须使用项）

2) read 接口中读取方式的修改

```
# read bytes, 运行pl_simulate的时候用这段代码, 直接使用read和pl_simulte.py读出来的结果是不正确的
# data = []
# for i in range(len):
#     data.append(map_.read_byte())
# data = np.array(data, dtype=np.uint8)
# data.dtype=dtype    # 按dtype整理数据

# read, 和pl侧联调的时候用这段代码
data = map_.read(len)
data = np.frombuffer(data, dtype = dtype).copy()
```

原方法使用 read_bytes 单字节读取，耗时长，因此在神经网络测试中修改为 read 方法，可实现 4 字节同时读取。

（注：该修改存在一定的问题，若使用 pl_simulate.py 脚本进行测试会出现读取结果不正确的问题，因此若同学们希望使用 pl_simulate.py 需要将该部分修改为 read_bytes 的方法）

3) BRAM 类中 write 接口变化

```
def write(self, data, block_name: str, offset='default'):
    '''写入数据
        由于数据位宽32bit，因此最好以4的倍数Byte写入

        Args:
            data: 输入的数据
            block_name: BramConfig中配置的block_info的key值
            offset: 支持两种输入模式
                    1. str: BramConfig中配置的offset字典key值
                    2. int: 在block上的偏移量
    '''
```

和 1)中变化一致，提供灵活的块内起始地址偏移方式，此处不再赘述。

4.3 测试结果

测试结果中存在以下几个字段：

- 0/20：表示在测试 20 张图片中编号为 0 的图片（编号范围为 0 ~ 19）
- Inference time：单次神经网络前向推理的时间
- Average time：平均每次神经网络前向推理的时间
- Accuracy：识别准确率

MLP 测试结果：

```
[INFO] test_mlp.py:36 - 14/20
[INFO] test_mlp.py:45 - Inference time: 0.012449264526367188
[INFO] test_mlp.py:46 - Average time: 0.012841018040974934
[INFO] test_mlp.py:36 - 15/20
[INFO] test_mlp.py:45 - Inference time: 0.01243734359741211
[INFO] test_mlp.py:46 - Average time: 0.012815788388252258
[INFO] test_mlp.py:36 - 16/20
[INFO] test_mlp.py:45 - Inference time: 0.01245260238647461
[INFO] test_mlp.py:46 - Average time: 0.01279442450579475
[INFO] test_mlp.py:36 - 17/20
[INFO] test_mlp.py:45 - Inference time: 0.012428760528564453
[INFO] test_mlp.py:46 - Average time: 0.012774109840393066
[INFO] test_mlp.py:36 - 18/20
[INFO] test_mlp.py:45 - Inference time: 0.012490510940551758
[INFO] test_mlp.py:46 - Average time: 0.012759183582506682
[INFO] test_mlp.py:36 - 19/20
[INFO] test_mlp.py:45 - Inference time: 0.012410640716552734
[INFO] test_mlp.py:46 - Average time: 0.012741756439208985
[INFO] test_mlp.py:51 - Accuracy: 95.0%
[INFO] test_mlp.py:52 - Average time: 0.012741756439208985
```

LeNet 测试结果：

```
[INFO] test_lenet.py:35 - 14/20
[INFO] test_lenet.py:44 - Inference time: 0.03456377983093262
[INFO] test_lenet.py:45 - Average time: 0.034726397196451826
[INFO] test_lenet.py:35 - 15/20
[INFO] test_lenet.py:44 - Inference time: 0.034575462341308594
[INFO] test_lenet.py:45 - Average time: 0.03471696376800537
[INFO] test_lenet.py:35 - 16/20
[INFO] test_lenet.py:44 - Inference time: 0.03456473350524902
[INFO] test_lenet.py:45 - Average time: 0.034708009046666766
[INFO] test_lenet.py:35 - 17/20
[INFO] test_lenet.py:44 - Inference time: 0.0345606803894043
[INFO] test_lenet.py:45 - Average time: 0.034699824121263295
[INFO] test_lenet.py:35 - 18/20
[INFO] test_lenet.py:44 - Inference time: 0.03455018997192383
[INFO] test_lenet.py:45 - Average time: 0.03469194863971911
[INFO] test_lenet.py:35 - 19/20
[INFO] test_lenet.py:44 - Inference time: 0.03506207466125488
[INFO] test_lenet.py:45 - Average time: 0.0347104549407959
[INFO] test_lenet.py:50 - Accuracy: 100.0%
[INFO] test_lenet.py:51 - Average time: 0.0347104549407959
```

5 实验内容

5.1 实验步骤

FPGA 侧：

1) 烧写 Lab5 中的 bitstream 至开发板

ARM 侧：

1) 替换 model/operators.py 中的 Matmul 类为自己的 Matmul 实现（该实现在 Lab4、Lab5 中均有用到）
2) 补全 model/mlp.py 以及 model/lenet.py 中网络定义以及前向推理部分（注意，ReLU 激活函数不需要实现，直接跳过即可，因为在量化过程中已经被融合）
3) 运行 test_mlp.py 以及 test_lenet.py，完成神经网络测试

5.2 问题

1) 对比以下两个方案的执行时间，对比执行时间：

a) 在 ARM 上使用 Numpy 运行矩阵乘法，执行神经网络
b) 使用 FPGA 运行矩阵乘法，执行神经网络

请回答方案 a 和 b 哪一个执行时间更快？为什么？（详细阐述分析的思路、过程）

5.3 附加题

1) 能否优化使用 FPGA 运行神经网络的时间？若能，请提出具体的方案，最好可以实现并验证效果
2) 当 ARM 侧采用跳地址存储的方式实现 BRAM 写入时运行时间如何？给出对比结果并分析
3) 参考 Lab5 中的问题 4，当总线数据位宽为 64bit 或者 128bit 时，神经网络推理运行时间如何？给出对比结果并加以分析