神经网络处理单元开发套件

V1.2.0　2020-11

目录

一、 概述

NDK（Neural-network processing unit Development Kit）能够帮助用户将训练好的浮点神经网络模型转换成物奇 WQ5007 芯片平台的神经网络处理单元（NPU）所支持的网络结构，将浮点网络参数转成定点数，最后打包成二进制文件。

神经网络处理单元（NPU）专门用于加速神经网络常见的运算，具体包括：

- 全连接层；
- 二维卷积层，可以加偏置，支持不对称卷积核（卷积核长宽不相等），空洞卷积（dilated convolution）；
- 非线性激活函数，包含 sigmoid、tanh、relu、relu6、leaky_relu（即 Caffe 中 slope 非 0 的 relu 运算）；
- LogSoftmax 运算，等价于一个 Softmax 运算并将输出的概率值取 log；
- 二维池化运算，包括最大值池化和均值池化；
- 多张量输入/输出运算，包括多个张量合并（concat）、张量分割（slice）、以及多个张量逐点求和等；
- 通道混洗（channel shuffle）运算；
- 其它可以等效为上述运算的操作。

更为详尽的网络层类型列表以及对各种层参数的约束请查阅后续章节《层类型说明》。

NDK 工具包提供了浮点、定点性能评估等辅助模型转换的功能。若经过定点化后性能损失较大，用户还可以使用 NDK 提供的工具在芯片平台对定点参数的限制下，对定点模型参数的进行训练。

最后，用户只需将由定点模型导出的二进制文件下载到 flash、配合芯片 SDK 使用，就能够在物奇芯片平台上轻松地调用神经网络模型。

二、 软件运行环境

NDK 的运行要求 Python 3.6 以上环境且安装有 numpy 工具包（版本 1.16.3，其它差异不大的 numpy 版本也可以）。部分 NDK 功能的使用需要依赖特定框架或工具包：

- 若待导入的神经网络模型文件格式为 caffemodel，需要安装 Caffe 框架；
- 若待导入的神经网络模型文件为 pb 或 hdf5，需要安装 TensorFlow 框架（版本 1.13.1）；
- 若待导入的神经网络模型文件为 onnx，则需要安装 python 的 onnx 工具包；
- 定点模型训练工具需要依赖 TensorFlow 框架；

- 模型普通定点化工具只需要 numpy 即可使用，但若环境中安装有 TensorFlow/PyToch（PyTorch 版本 1.2.0），则能够通过配置加速选项大幅提高运行效率。
- 模型分析工具函数中，导出到 csv 文件功能需要依赖 pandas 工具包，如环境中没有安装 pandas 该导出功能将无法使用，分析结果只能以 python 字典的形式返回。

三、 NDK 数据类型

神经网络模型在 NDK 中使用成对的一个 Layer 对象列表和一个参数字典进行表示，其中 Layer 对象列表给出了网络的结构描述，而参数字典则存储了网络的权重、偏置值以及定点化参数。

在进行网络性能评估、分析等一些需要批量导入输入数据的时候，NDK 要求用户将输入数据以 Python 生成器函数（Generator）的方式给出，相应的函数在函数内部通过调用`next()`方法获取数据。

3.1 模型结构描述—Layer 类型对象列表

ndk.layers.Layer

在转换工具内部，我们参考 Caffe 框架中的 prototxt 定义了一个 Python 类 Layer，用以描述神经网络模型中包含的各个层。同时，NDK 对模型结构的描述比标准的 Caffe 框架更为严格，例如对卷积层和池化层涉及的填充操作，NDK 要求 Layer 对象明确地分别给出四个方向上的填充数量，且不支持自动填充（模型解析工具会根据实际情况自动计算这些参数）。

一个完整的神经网络通过一个由 Layer 对象组成的 Python 列表存储。

Layer 类型的对象至少包括以下四个属性：

1) 层类型（type）：字符串对象，如'Sigmoid'、'Convolution'、'InnerProduct'等。
2) 层名称（name）：字符串对象。用以描述一个模型的 Layer 对象列表中不允许有同名的两个层，在模型解析的时候需要检查和处理层同名的问题。
3) 输入张量名（bottom）：字符串对象，或字符串对象列表（有多个输入张量）
4) 输出张量名（top）：字符串对象，或字符串对象列表（有多个输出张量）

除此以外，不同的层类型具有不同的特有属性。

3.2 模型参数描述—参数字典

神经网络模型各个层的权重、偏置、以及定点化参数使用一个 Python 字典存储，其中键名和值类型作如下约定：

1) 浮点网络中的权重，键的格式为"层名称_weight"，值为浮点数 numpy.ndarray 对象；

2) 浮点网络中的偏置，键的格式为"层名称_bias"，值为浮点数 numpy.ndarray 对象；

3) 定点网络中的权重，键的格式为"层名称_quant_weight"，值为有符号整数 numpy.ndarray 对象；

4) 定点网络中的偏置，键的格式为"层名称_quant_bias"，值为有符号整数 numpy.ndarray 对象；

5) 定点网络中的权重的小数点位置，键的格式为"层名称_frac_weight"，值为有符号整数，或有符号整数 numpy.ndarray 对象（各通道定点权重值的小数点位置可以不同）；

6) 定点网络中的偏置的小数点位置，键的格式为"层名称_frac_bias "，值为有符号整数，或有符号整数 numpy.ndarray 对象（各通道定点偏置值的小数点位置可以不同）；

7) 定点网络中的特征张量的小数点位置（包含输入图像的小数位数），键的格式为"张量名_frac "，值为有符号整数；

8) 定点网络中的特征张量是否为有符号数，键的格式为“张量名_signed”，值为布尔数；注意，如果参数字典中没有该键，则对应张量的数值类型默认为有符号数。

其中，用整型变量q表示定点化后的权重/偏置，用整型变量n表示小数点位置，该定点数(q,n)对应的浮点值x表示如下：

$$x = q \times 2^{-n}$$

3.3 数据样本生成器

在进行定点化、定点模型训练或者定点模型误差分析时，相关操作都需要相当数量的数据样本用于训练或评估。

因此，相关定点化操作以及分析的函数均要求用户在外部编写一个数据样本生成器（generator）作为函数的输入，函数内部通过调用该生成器产生足够数量的数据样本。

数据样本生成器是一个 Python 的生成器函数（对 generator 不熟悉的用户请参考后续章节《Python 生成器用法简介》），要求每次对该生成器调用`next()`方法时，返回一个 Python 字典类型的数据，其中包含 1 或 2 个键：

- “input”键 – 神经网络模型的输入数据，必须存在。
- “output”键 – 神经网络模型的输出参考数据，在做普通定点化时允许不存在。

它们的值分别对应一个 numpy.ndarray 类型的 4 维数组，格式是 NCHW，四个维度含义依次分别是 Batch 大小、通道数、高度、宽度。

注意，生成器返回的数据中“input”和“output”是指神经网络的输入、输出：如有所有预处理操作都应该在该生成器返回数据之前完成；如有后处理操作则应该将数据还原成网络输出对应的值。

鉴于在训练或者性能评估过程中需要的样本数不确定，建议通过设定死循环（`while True`）确保相关函数可以通过该生成器获得无限多的数据样本。

例如，读取 MNIST 数据时，首先需要将原数据集中的 data 经过必要的归一化处理（如果网络模型的输入要求为 0-1 之间的浮点数），然后转成 NCHW 格式后赋值给“input”键；而“output”则需要将对应的 label 处理成 one-hot 向量并且额外扩充 HW 两个大小为 1 的维度。

四、 NDK 函数集

NDK 工具函数包括模型解析工具、模型优化工具、模型定点化工具、误差分析工具、模型打包工具以及其它辅助工具函数。这些函数大致分为三类操作：模型转换工具、模型打包工具和性能分析工具。

模型转换工具能够将一个用户给定的浮点模型解析后导入，然后经过层合并、定点化等操作转换成 NPU 能够支持的网络结构和定点权重/偏置值。模型转换工具可以将转后的网络结构保存在一个 prototxt 文件里，对应的权重/偏置值则保存在 npz 文件。

模型打包工具读取 prototxt 和 npz 文件，将转换工具输出的网络和权重/偏置值打包成 bin 文件。将生成的 bin 文件下载到 flash 后，配合芯片的 SDK 能够在物奇芯片平台上方便地调用神经网络模型。

性能分析工具则可以帮助用户分析浮点、定点模型中各个张量的分布情况以及比较定点化前后模型里各个张量的差异。用户也可以通过调用其它辅助函数搭建脚本，满足自定义的分析需求。

4.1 模型解析工具

目前，NDK 支持对 Caffe 和 TensorFlow 两种框架下模型文件的解析。解析工具可以将成对的 prototxt 和 caffemodel 文件、或 pb 文件转换成 NDK 所需的 Layer 列表和参数字典，以供后续定点化等处理使用。

4.1.1 Caffe 框架模型解析

ndk.caffe_interface.load_from_caffemodel

在 Caffe 框架下将待导入的神经网络模型保存为一对.prototxt 和.caffemodel 文件，作为解析函数的输入。

解析函数从.prototxt 文件中读取神经网络结构信息、从.caffemodel 文件中读取对应的网络参数字典。

解析后的模型结构保存在一个 Layer 类型对象的列表里，对应的权重/偏置保存为 Python 字典类型的参数字典，作为解析函数的输出。

NDK 要求作为输入的 prototxt 的第一个 layer 必须为 Input 类型，指定网络输入的各个维度大小，如果不符合这个要求，请手动在 prototxt 中添加这一层。Input 层在 prototxt 文件中的格式请参见《层类型说明》的相关小节。

解析 Caffe 模型的接口函数调用形式如下：

```
layer_list, param_dict = load_from_caffemodel(
                                    fname_prototxt,
                                    fname_caffemodel
                                    )
```

必要参数：

- **fname_prototxt** – 字符串类型，待导入模型对应的.prototxt 文件路径
- **fname_caffemodel** – 字符串类型，待导入模型对应的.caffemodel 文件路径

返回参数：

- **layer_list** – Layer 对象的 Python 列表类型，解析所得模型的网络结构信息
- **param_dict** – Python 字典类型，解析所得模型的参数字典

4.1.2 TensorFlow 框架模型解析

TensorFlow 框架下将待导入的神经网络模型保存为.pb 或者.hdf5 文件，作为解析函数的输入。

ndk.tensorflow_interface.load_from_pb

解析函数从给定的.pb 文件中读取神经网络结构信息和网络参数字典。

解析后的模型结构保存在一个 Layer 类型对象的列表里，对应的权重/偏置保存为 Python 字典类型的参数字典，作为解析函数的输出。

解析 TensorFlow 模型的接口函数调用形式如下：

```
layer_list, param_dict = load_from_pb(fname_pb)
```

必要参数：

- **fname_pb** – 字符串类型，待导入模型对应的.pb 文件路径。

返回参数：

- **layer_list** – Layer 对象的 Python 列表类型，解析所得模型的网络结构信息
- **param_dict** – Python 字典类型，解析所得模型的参数字典。

ndk.tensorflow_interface.load_from_hdf5

解析函数从给定的. hdf5 文件中读取神经网络结构信息和网络参数字典。

解析后的模型结构保存在一个 Layer 类型对象的列表里，对应的权重/偏置保存为 Python 字典类型的参数字典，作为解析函数的输出。

解析 TensorFlow 模型的接口函数调用形式如下：

```
layer_list, param_dict = load_from_hdf5(fname_hdf5)
```

必要参数：

- **fname_hdf5** – 字符串类型，待导入模型对应的.hdf5 文件路径。

返回参数：

- **layer_list** – Layer 对象的 Python 列表类型，解析所得模型的网络结构信息
- **param_dict** – Python 字典类型，解析所得模型的参数字典。

4.1.3 ONNX 格式模型解析

Open Neural Network Exchange（ONNX，开放神经网络交换）格式，是一个用于表示深度学习模型的标准，可使模型在不同框架之间进行转移。

读取 ONNX 文件前，请检查 python 环境中是否已安装有 onnx 工具包。如若不然，请先使用命令 `pip install onnx` 进行 onnx 工具包的安装。

ndk.onnx_interface.load_from_onnx

解析函数从给定的.onnx 文件中读取神经网络结构信息和网络参数字典。

解析后的模型结构保存在一个 Layer 类型对象的列表里，对应的权重/偏置保存为 Python 字典类型的参数字典，作为解析函数的输出。

解析 ONNX 格式文件的接口函数调用形式如下：

```
layer_list, param_dict = load_from_onnx(fname_onnx)
```

必要参数：

- **fname_onnx** – 字符串类型，待导入模型对应的.onnx 文件路径

返回参数：

- **layer_list** – Layer 对象的 Python 列表类型，解析所得模型的网络结构信息
- **param_dict** – Python 字典类型，解析所得模型的参数字典

4.2 模型优化工具

模型优化模块提供自动扫描、优化工具，以提高神经网络模型在芯片平台上的执行效率。

4.2.1 层合并工具

ndk.optimize.merge_layers

对一些层类型进行等效的合并，具体来说，是将 Scale、Bias 和 BatchNorm 层合并到前/后的 Convolution 或者 InnerProduct 层里。

函数输入为原模型对应的 Layer 对象列表和参数字典。模块输出为层合并后所得模型的 Layer 对象列表和参数字典。

层合并操作 1：BatchNorm 和 Scale 收进前面一层卷积。

在推理过程中，BatchNorm 层学习到的均值和方差都可以视为和某一个 feature 无关的常数。设u_c和v_c分别为 BatchNorm 操作里的均值和方差（带下标c是因为不同通道的均值和方差可能不同）。

设X为卷积的输入、Y为卷积的输出也就是 BatchNorm 的输入、Z为 BatchNorm 的输出、W为权重，有：

$$Y_{c,x,y} = \sum_{i,h,w} X_{i,x+h,y+w} W_{c,i,h,w}$$

$$Z_{c,x,y} = \frac{Y_{c,x,y} - u_c}{\sqrt{v_c + \varepsilon}} = \frac{\sum_{i,h,w} X_{i,x+h,y+w} W_{c,i,h,w} - u_c}{\sqrt{v_c + \varepsilon}} = \sum_{i,h,w} X_{i,x+h,y+w} \frac{W_{c,i,h,w}}{\sqrt{v_c + \varepsilon}} - \frac{u_c}{\sqrt{v_c + \varepsilon}}$$

$\sum_{i,h,w} X_{i,x+h,y+w} \frac{W_{c,i,h,w}}{\sqrt{v_c+\varepsilon}}$也是一个卷积结果，该卷积的权重为$\frac{W_{c,i,h,w}}{\sqrt{v_c+\varepsilon}}$。

如果该 BatchNorm 后面级联了 Scale，那么 Scale 也可以按照同样的方式合并入前面的卷积。

层合并操作 2：BatchNorm 和 Scale 收进后面一层卷积。

设 Scale 运算中的乘数为γ，偏置为β。设卷积运算的权重为W，偏置为b。设 Scale 输入为X, Scale 输出也就是卷积输入为Y, 卷积输出为Z。于是有(这里假设卷积没有 padding)：

$$Y_{i,h,w} = X_{i,h,w}\gamma_i + \beta_i$$

$$Z_{c,h,w} = \sum_{i,x,y} Y_{i,x+h,y+w} W_{c,i,x,y} + b_c = \sum_{i,x,y} (X_{i,x+h,y+w}\gamma_i + \beta_i) W_{c,i,x,y} + b_c$$

$$= \sum_{i,x,y} X_{i,x+h,y+w}\gamma_i W_{c,i,x,y} + \sum_{i,x,y} \beta_i W_{c,i,x,y} + b_c$$

因此，从X到Z可以视为一个卷积运算，其权重为$\gamma_i W_{c,i,x,y}$，偏置为$\sum_{i,x,y} \beta_i W_{c,i,x,y} + b_c$。

注意观察上式：当卷积层有填充（padding）时，该过程相当于是先对X进行了填充，然后才顺序进行 Scale 运算和卷积运算。

因此，该方式在卷积层有填充且 Scale 层的偏置非 0 会导致与 padding 相关的外围一圈元素计算结果略有误差（由于卷积 padding 值为 0，合并后权重项不受影响），所以合并时请先评估该误差对整个网络运行结果的影响。

对神经网络层进行等效合并的接口函数调用形式如下：

```
merged_layer_list, merged_param_dict = merge_layers(
                                        layer_list,
                                        param_dict,
                                        fname_log
                                        )
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，待合并的网络模型结构信息；
- **param_dict** – Python 字典类型，待合并的网络模型对应的参数字典。

可选参数：

- **fname_log** – 字符串类型，默认值为 None，若配置了该参数，在执行层合并操作之后会将合并操作日志保存到指定文件。

返回参数：

- **merged_layer_list** – Layer 对象的 Python 列表类型, 层合并后模型的网络结构信息；
- **merged_param_dict** – Python 字典类型，层合并所得模型的参数字典。

注意，网络层合并操作仅针对浮点网络。因此，建议在进行定点化操作之前，先进行层合并操作。

4.2.2 预处理工具

ndk.optimize.add_pre_norm

大多数神经网络要求将原始图像归一化，使用 NPU 完成该操作可以明显缩短耗时。因此建议使用该工具将归一化运算并入神经网络中。并入归一化运算的神经网络，使用原始图像数据作为输入。

将归一化运算并入神经网络的接口函数调用形式如下：

```
add_pre_norm (layer_list, param_dict, weight, bias)
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息；
- **param_dict** – Python 字典类型，网络模型对应的参数字典；
- **weight** – 浮点数 numpy.ndarray 对象，归一化运算中，对原始图像每个通道相乘的系数；
- **bias**– 浮点数 numpy.ndarray 对象，归一化运算中，对原始图像每个通道相加的系数。

特别的，如果对图像的归一化运算为：

$$(x - mean)/std$$

则 weight 参数取 1/std，bias 参数取-mean/std。

4.3 模型定点化工具

模型定点化模块帮助用户将一个浮点模型转成符合芯片平台要求的定点模型。

NDK 工具提供了普通定点化和定点化训练两种模型定点化转换方式：

1) 普通定点化方式通过统计模型权重/偏置、特征张量的分布特性来决定定点参数，允许用户配置不同的优化目标（如根据动态范围、最小化均方误差、最小化 KL 散度）；
2) 定点化训练方式则首先通过模型的权重/偏置/特征张量的动态范围决定定点参数，然后通过训练过程对模型权重/偏置进行调整。

4.3.1 普通定点化操作

ndk.quantize.quantize_model

输入需要 Layer 列表和参数字典之外，还需要一个用以统计数据分布的测试集合，以 Python 生成器函数的形式给出。定点化函数的输出为定点化后更新的 Layer 列表和对应的包含定点参数的参数字典。

如果测试集合中的数据在给到网络之前需要进行预处理，需要用户在编写生成器函数的时候就处理完：要求对生成器函数调用 `next()` 方法得到的结果中，“input”键对应的数组是神经网络一个 batch 的输入样本。

此外由于芯片平台对一些层的定点化参数有一定的限制，在定点化过程中遇到这些情况，本函数会综合考虑多个层，确保输出的参数符合芯片平台的要求。

比如卷积层后接 ReLU 层的情况：ReLU 层要求输入输出张量的定点参数一致；而由于 ReLU 层的存在，卷积层输出张量中小于零的值会被统一压缩到零、这些负值的绝对值大小对后续层并无影响，没有必要为了维持这些负值的精度。因此定点化操作会根据 ReLU 层输出张量的分布来确定卷积层的输出小数点位置。

需要特别提醒的是 LogSoftmax 层：由于芯片平台中 LogSoftmax 层的输入张量小数点位置是确定的，也就是说输入张量会被饱和到区间(-8, 8)，具体参考《层类型说明》章节。为了保证输入张量中的最大值不至于被饱和，定点化函数会尝试在 LogSoftmax 前面的 Convolution/InnerProduct 层的偏置上统一加减一个数。但是在某些情况下，这样增加偏置值的作法可能使得前一层的偏置定点参数变得难以合理设定（例如在原本动态范围比较小的偏置值 A 上再统一加上了一个较大的数值 B，在 8/16 比特定点化时由于精度损失，会让原本的偏置值 A 被“损失”掉而只保留 B，从而导致原本的偏置值 A 被无效化了）。因此建议用户将 LogSoftmax 层删除后再调用定点化函数，然后尝试利用性能分析工具查看相关张量的分布后手动调整最后几层的定点参数以适配 LogSoftmax 的输入范围限制，或者直接将最后一层 Convolution/InnerProduct 的输出张量交给 CPU 进行浮点的 LogSoftmax 运算。

对神经网络层进行普通定点化操作的接口函数调用形式如下：

```
quant_layer_list, quant_param_dict = quantize_model(
                                     layer_list,
                                     param_dict,
                                     bitwidth,
                                     data_generator,
                                     usr_param_dict=None
                                     method_dict=None,
                                     aggressive=True,
                                     gaussian_approx=False
                                     factor_num_bin=4,
                                     num_step_pre=20,
                                     num_step=100,
                                     drop_logsoftmax=True,
                                     priority_mode='fwb',
                                     compensate_floor_on_bias=True,
                                     log_dir=None
                                     )
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，待定点化的网络模型结构信息；
- **param_dict** – Python 字典类型，待定点化的网络模型对应的参数字典；
- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16；
- **data_generator** – Python 生成器类型，给出定点化过程中用作统计各网络层输入输出数据分布的测试用的数据样本，每次调用 `next()`时返回字典数据中必须包含模型的输入样本（即'input'键），可以不含模型的输出样本。

  如前所述，输入和输出样本均要求是一个 4 维 numpy.ndarray 对象，按照 NCHW 排列，所以定点化时使用数据的 batch 大小由该输入数组对象的第一维决定。

可选参数:

- **usr_param_dict** – Python 字典类型，用户指定的定点化参数字典，默认为 None，表示一个空字典。

  用户可以通过这个字典直接对部分张量的定点参数进行指定。

  如果配置了该参数，本函数会尝试读取其中的"层名称_weight_frac"、"层名称_bias_frac"以及"张量名_frac"这三种格式的字段。在遇到对应层的权重、偏置或者特征张量的时候，将会忠实地根据指定的小数点位置进行操作；否则小数点位置根据统计各个张量的分布确定小数点位置。

- **method_dict** – Python 字典类型，默认值为 None，表示一个空字典。

  该字典的键名为张量名。若为特征张量，则直接使用该特征张量的"张量名"；若为某层的权重或者偏置，则为"层名称_weight"或"层名称_bias"；

  对应的值可以为"MINMAX"、"MSE"、或者"KL"，分别表示采用相应策略对指定张量进行定点化:

  - MINMAX - 按照动态范围（统计得到的最大、最小值）确定定点参数

    对于张量X，若为有符号数，则其小数点位置为

    $$\mathrm{X\_frac} = \left\lfloor -\log 2\left(\frac{\max(|X|)}{2^{b-1}-1}\right)\right\rfloor$$

    若为无符号数，则其小数点位置为

    $$\mathrm{X\_frac} = \left\lfloor -\log 2\left(\frac{\max(X)}{2^{b}-1}\right)\right\rfloor$$

    其中，b为定点数位宽，运算$\lfloor\cdot\rfloor$为向下取整。

  - MSE – 根据最小化均方误差（MSE）准则确定定点参数

    定点化函数通过计算张量X的浮点值x和不同定点参数（由于定点数位宽已由用户指定，此处仅指小数点位置）对应的定点值$\mathrm{Q}(x,n)$之间的均方误差，

    $$\mathrm{MSE}(X,n) = \mathrm{AVG}([x - Q(x,n)]^2)$$

    其中$\mathrm{AVG}(\cdot)$为求平均运算。

定点化函数比较各个配置下的 MSE，然后确定张量X的小数点位置n为：能满足芯片限制并且使得 MSE 最小。

- KL – 根据最小化 Kullback-Leibler (KL)散度的准则确定定点参数

  KL 散度常用于度量两个分布之间的相似性：KL 散度值越大表明两个分布的差异越大；反之，KL 散度值越小表明两个分布的差异越小。

  当某张量的定点化方法被指定为“KL”时，定点化函数通过统计该张量的浮点值概率分布$\mathrm{p}(x)$以及在不同定点参数（由于定点数位宽已由用户指定，此处仅指小数点位置）对应的定点值概率分布$\mathrm{q}(x,n)$，计算 KL 散度

$$\mathrm{KLD}(X,n)=\sum p(x)\log\frac{p(x)}{q(x,n)}$$

  定点化函数计算并比较各个配置下的 KL 散度，然后确定张量X的小数点位置n为：能满足芯片限制并且使得 KLD 最小。

默认情况下，即某张量的名称没有出现在 `method_dict` 中，则按照 MINMAX 准则确定。

例如，要对名为“conv1”层的权重使用最小化 KL 准则、对偏置使用 MSE 准则进行定点化，则需要在 `method_dict` 中添加`{'conv1_weight': 'KL', 'conv1_bias': 'MSE'}`；进一步地，若要对该层的输出特征张量“conv1_out”使用 MINMAX 准则进行定点化，则需要在 `method_dict` 中添加`{'conv1_out': 'MINMAX'}`，也可以什么都不做、同时确保 `method_dict` 中没有名为'conv1_out'的键。

- **aggressive** – 布尔类型，默认值为 True。

  若设置为 True，则首先在浮点模型的基础上运行 `num_step_pre` 个 batch 统计各层的权重/偏置并估计各个特征张量的大致动态范围，根据各个张量的动态范围确定统计张量概率分布用的 bin 的中心值，然后通过 `num_step` 个 batch 统计各个张量的分布，并确定定点参数。

  若设置为 False，则会逐层进行定点参数的确定：首先在浮点模型的基础上运行 `num_step_pre` 个 batch 统计各层的权重/偏置并估计各个特征张量的大致动态范围，根据各个张量的动态范围确定统计张量概率分布用的 bin 的中心值；然后根据 `num_step` 个 batch 统计模型输入张量的分布，并确定点参数；在这之后之后，根据定点化后的模型输入和第一层定点化后的权重/偏置，通过另外 `num_step` 个 batch 对第一层的输出特征张量的数值分布进行统计，并确定第一层输出特征张量的定点参数；根据第一层输出的定点参数作为输入，统计第二层的输出特征张量的分布并确定定点参数……逐层进行，直至所有层的定点参数均被确定。

- **gaussian_approx** – 布尔类型，默认值为 False。

  在 `aggressive=False`（逐层确定定点参数）时，当前待定点化层的输入特征张量需要通过执行前面所有已经定点层得到。因此在定点化靠后的层之前，最开始的层会被反复多次地执行，从而影响执行效率。

  通过设定 `gaussian_approx=True`，定点化函数会用高斯分布近似特征张量的分布，每一个张量只需要记录对应的均值和方差。在需要产生该特征张量的值时，根据均

值和方差产生随机张量，从而避免反复执行网络中前置的层，从而提高定点化函数执行效率。

但若某特征张量的真实分布与高斯分布相去甚远，则可能会对最终的定点模型性能造成影响。

本参数自 NDK-1.1.0 版开始暂停支持，当前版本配置后不起作用。

- **factor_num_bin** – 整数，默认值为 4，在依据最小化 MSE 或者 KL 准则确定定点参数时需要统计数据分布，bin 的数量由定点数位宽和 `factor_num_bin` 共同决定:

$$\text{num\_bin} = 2^{\text{bitwidth+factor\_num\_bin}}$$

- **num_step_pre** - 整数, 默认值 20, 通过将 `num_step_pre` 个 batch 送入浮点模型，得到各个张量的大致动态范围。

- **num_step** - 整数，默认值 100，通过 `num_step` 个 batch 确定模型特征张量的定点参数，或逐层用 `num_step` 个 batch 确定当前层的定点参数。

- **drop_logsoftmax** – 布尔型，默认值为 True。

  由于 LogSoftmax 层在硬件中限制比较严格（参见《层类型说明》），这会显著影响定点化性能效果。

  当设置为 True 时，定点化函数会自动删除模型最后的 LogSoftmax 层，并打印一个警告信息。请用户在定点化结束后，通过 CPU 去计算 LogSoftmax/Softmax 层或者手工调整该层参数。

  当设置为 False 时，定点化函数会首先将 `num_step_pre` 个 batch 送入浮点模型，统计 LogSoftmax 层输入张量的动态范围，并额外加一个常数到 LogSoftmax 之前一层的 Convolution/InnerProduct 层的偏置项，以尽量避免 LogSoftmax 层由于输入数值被饱和。

- **priority_mode** – 字符串类型。由于硬件对每一层输入输出张量、权重、偏置的小数点位置配置有限制，优先级低的参数在确定时更可能因受到硬件限制的关系而被设置为一个次优解。

  六种优先顺序的名称如下，用户可以通过配置本参数改变优先顺序，默认值'fwb':

  'fwb' – 对每一层，首先确定输出张量的小数点位置，然后是权重，最后是偏置;

  'fbw' – 对每一层，首先确定输出张量的小数点位置，然后是偏置，最后是权重;

  'wfb' – 对每一层，首先确定权重的小数点位置，然后是输出张量，最后是偏置;

  'wbf' – 对每一层，首先确定权重的小数点位置，然后是偏置，最后是输出张量;

  'bfw' – 对每一层，首先确定偏置的小数点位置，然后是输出张量，最后是权重;

  'bwf' – 对每一层，首先确定偏置的小数点位置，然后是权重，最后是输出张量;

- **compensate_floor_on_bias** – 布尔型，默认值 True。硬件计算在遇到需要截断时都是采用的向下取整的方式。相比四舍五入的方式，隐藏层的特征张量在精度外可能会有一个微小的向负半轴的误差。

本参数设置为 True，则定点化函数尝试在各个有偏置项的层上增加一个额外值，尽可能地将该层输出张量补偿为四舍五入。

本参数设置为 False，则定点化函数不作任何特别的动作。

- **log_dir** – 字符串类型，设置日志文件存放路径，默认值 None。

  配置了本参数之后，定点化函数会记录定点化过程中，各个定点参数的执行依据，并将日志文件保存到指定路径下。

  配置成 None 时，本函数不产生任何日志文件；

返回参数：

- **quant_layer_list** – Layer 对象的 Python 列表类型，定点模型的网络结构信息；
- **quant_param_dict** – Python 字典类型，定点化所得模型的参数字典。

4.3.2 Tensorflow 定点模型训练

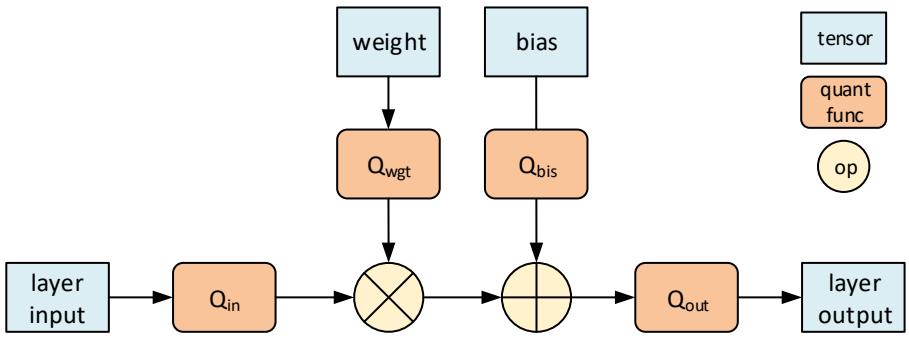
ndk.quantize.quantize_model_with_training

本功能需要环境中装有 TensorFlow。

若模型在经过普通定点化后性能损失较大（进行 8 比特模型定点化的时候更容易出现这种情况），NDK 还提供了根据定点化参数重新进行网络权重/偏置训练的工具。

本质上说，这是一个迁移训练的过程：在浮点网络的相应位置插入定点化处理节点（相当于激活函数），然后在这个经过定点化的计算图上重新训练神经网络。

以全连接层为例，定点模型训练函数会根据 Layer 类型的 Dense 层构建一个 TensorFlow 的 dense 层（由 MatMul 和 Add 两个 OP 构成），并且在 MatMul 操作的输入、Add 操作的输出、权重张量和 MatMul 之间、偏置张量和 Add 之间增加定点化处理节点。



定点化处理节点的前向计算是标准的定点化操作：

$$Q(x, \text{bitwidth}, \text{frac}) = \Delta \cdot \text{floor}\left(\frac{\text{clip}(x, x_{min}, x_{max})}{\Delta}\right)$$

其中$\Delta = 2^{\text{frac}}$，$x_{max} = -x_{min} = \Delta \cdot \left(2^{\text{bitwidth}-1} - 1\right)$，函数$\text{floor}(\cdot)$表示下取整，截断函数$\text{clip}(x, x_{min}, x_{max}) = \max(\min(x, x_{max}), x_{min})$。

由于以上函数在各个量化区间的边缘处不可导，在进行梯度传播时需要使用一个近似计算方法。本函数中定点化处理节点的后向计算按下式进行：

$$\frac{\partial Q}{\partial x} = \begin{cases} 1 & if\ x_{min} \le x \le x_{max} \\ 0 & \text{otherwise} \end{cases}$$

定点模型训练函数除了需要一个用于描述原浮点模型的 Layer 列表之外（如果提供了参数字典则会读取其中的浮点参数作为初始权重/偏置，如果没有提供参数字典则从随机权重/偏置开始训练），还需要一个用以统计数据分布的测试集合（需要网络输入数据样本“input”和与其对应的目标网络输出样本”output”）。

定点模型训练函数的输出为定点化后更新的 Layer 类型列表和对应的参数字典。

对神经网络层进行定点训练的接口函数调用形式如下：

```
quant_layer_list, quant_param_dict = \
    quantize_model_with_training( layer_list,
                                  bitwidth,
                                  param_dict,
                                  data_generator=None,
                                  input_fn=None,
                                  log_dir='log',
                                  loss_fn=None,
                                  optimizer=None,
                                  num_step_train=1000,
                                  num_step_log=100,
                                  perlayer_quant_layer_names=None,
                                  layer_group=1,
                                  rnd_seed=None)
```

特别说明：

- 本函数是对神经网络进行重新训练，如果模型比较大，对电脑配置（如内存、GPU 等）的要求会比较高；
- LogSoftmax 层在硬件中限制比较严格（参见《层类型说明》），这会严重影响训练效果。因此如果网络的输出是 LogSoftmax 层，请在保存模型时或者在导出的 prototxt 文件中将其删除后再进行定点模型训练；训练结束后，通过 CPU 去计算 LogSoftmax/Softmax 或者手工调整该层参数。

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，待定点化的网络模型结构信息；
- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16；
- **param_dict** – Python 字典类型，待定点化的浮点网络模型对应的参数字典。

  在定点模型训练过程开始之前，会首先用 `param_dict` 中各层对应的浮点权重和偏置值（即“层名称_weight”和“层名称_bias”字段）进行初始化；

然后会尝试从 `param_dict` 中读取各层小数点位置信息(即 “层名称_weight_frac”、“层名称 _bias_frac”以及“张量名_frac”这三种格式的字段),并在相应位置插入定点操作节点。

如果小数点位置信息有缺失，则会自动调用普通定点化模块生成这些信息。

可选参数:

- **data_generator** – python 生成器类型，默认值为 None。每次调用 `next()` 时返回字典数据中必须包含模型的输入样本(即'input'键), 以及模型的输出样本(即'output'键）；

  如前所述, 输入和输出样本均要求是一个 4 维 numpy.ndarray 对象, 按照 NCHW 排列，所以训练时的 batch 大小由该输入/输出数组对象的第一维决定。`data_generator` 参数和 `input_fn` 参数中必须有一个不为 None。

- **input_fn** – 输入函数, 默认值为 None。调用该输入函数返回一个 tf.data.Dataset 对象或者一个元组。这个元组为 (features, labels)。features 为网络的输入特征张量, shape 为[batch_size, channels, height, width]，labels 为跟网络输入对应的标签。如果调用 `input_fn` 返回一个 tf.data.Dataset 对象，则通过此 Dataset 获取的值为一个元组，该元组跟输入函数返回一个元组有同样的要求。`data_generator` 参数和 `input_fn` 参数中必须有一个不为 None。

- **log_dir** – 字符串对象，日志目录，默认值为'log'。

  用户可以使用 TensorBoard 工具对日志数据进行可视化分析。

  定点训练过程中还需要保存一些中间模型的参数，这些中间结果也会保存在 `log_dir` 所指定的目录下。

- **loss_fn** – TensorFlow 张量操作函数，定义了在定点训练过程中需要的损失函数，默认为 None。

  如果为 None，则自动初始化为 `tf.losses.sparse_softmax_cross_entropy`。

- **optimizer** – tf.train.Optimizer 类型的对象，默认为 None。

  如果传入值为 None，则自动初始化为 `tf.train.AdamOptimizer()`，根据 TensorFlow 文档默认学习率为 0.001；若需使用其它优化方法以及修改学习率等参数，请在调用本函数之前创建优化器对象。

- **num_step_train** – 整数，默认值 1000。

  `num_step_train` 指定了训练每一层（组）时所执行的训练步数。

- **num_step_log –** 整数类型，默认值为 100。训练时每隔 `num_step_log` 在屏幕上打印一条 log，并将训练 loss 和准确率写入 TensorFlow 的 event file。可以通过 TensorBoard 进行查看。

- **perlayer_quant_layer_names**: 字符串集合, 包含有层名的列表, 默认值为 None (表示空集合）。

默认情况下，即某一层名没有包含在 `perlayer_quant_layer_names` 列表中，对该层进行定点化时采用逐通道的方式，即其权重/偏置（如果有的话）对应每个输出通道都分别有一组独立的定点参数；

若某一层的层名包含在 `perlayer_quant_layer_names` 列表中，对该层进行定点化时采用逐层的方式，即其权重/偏置（如果有的话）对应各个输出通道共享同一组定点参数。

设置某一层为逐层方式进行定点化，相比逐通道方式增加了额外的限制，在一定程度上会损失性能，但这样做可以减少插入的定点化处理节点，提高定点训练执行速度。

- **layer_group**：整数，必须大于 0，默认值为 1。

  `layer_group` 可能会影响最终的模型性能。设定一个合适的 `layer_group` 能在不影响最终模型性能的情况下，有效减短定点训练的执行时间；

  当 `layer_group` 大于网络总层数，则进行全模型定点训练。全模型定点训练的流程为：对所有层插入定点化处理节点，然后对整个网络进行训练、同步调整所有层的权重/偏置。

  当 `layer_group` 大于等于 1，且小于网络总层数时，进行逐层（组）定点训练。逐层（组）定点训练的流程为：根据参数 `layer_group` 的值（默认值为 1）进行分组：第 1 组为第一层（一定是 Input 层）和之后的 `layer_group` 层，第 2 组为再之后的 `layer_group` 层……训练最开始仅仅对第 1 组的层插入定点处理节点，就启动对网络的训练，调整权重和偏置值；然后，对第 2 组的层差插入定点处理节点，固定第 1 组的权重/偏置为不可训练，对其余层进行训练（如果其余的层都不可训练，则自动跳过训练过程）……每一轮训练对当前组插入插入定点处理节点，并且仅允许对当前组和所有后续组的权重和偏置进行调整，直至完成所有层的训练。

  当网络总层数不能被 `layer_group` 整除时，最后一组为剩下的层。例如当网络总层数为 6，`layer_group` 为 2，则将第 1、2、3 层分为一组（包含 Input 层），第 4、5 层为一组，第 6 层为一组。

- **rnd_seed** – 整数，默认值为 None。如果设置了该值，则会在训练之前用这个数作为随机种子，用于需要重现训练结果的情况。

返回参数：

- **quant_layer_list** – Layer 对象的 Python 列表类型，定点模型的网络结构信息；
- **quant_param_dict** – Python 字典类型，定点模型的参数字典。

4.3.3 Pytorch 定点模型训练

NDK 支持将普通定点化得到的 NDK 网络模型转换成 pytorch 框架的自定义模型，用户可以使用该模型，按照 pytorch 框架要求的方式进行定点训练。

```
ndk.quant_train_torch.quant_layer.QuantizedNet
```

该类继承了 torch.nn.Module 类，因此用户可以将其视为一个 pytorch 自定义模型使用。

该类构造函数参数列表如下：

```
QuantizedNet( bitwidth,
              layer_list,
              param_dict,
              quant_output = True,
              quant_weight = True,
              quant_bias = True,
              hw_aligned = True,
              bitwidth = 8,
              for_training = True
              )
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，待保存的神经网络模型结构信息；
- **param_dict** – Python 字典类型，待保存的网络模型对应的参数字典；

可选参数：

- **quant_output** – 是否量化网络每一层的输出，默认为 True，用户在进行定点训练时使用默认参数即可。为 True 时，需要 param_dict 中包含网络所有特征张量的小数点位置。
- **quant_weight** – 是否量化网络每一层的权重，默认为 True，用户在进行定点训练时使用默认参数即可。为 True 时，需要 param_dict 中包含定点权重和小数点位置。
- **quant_bias** – 是否量化网络每一层的偏置，默认为 True，用户在进行定点训练时使用默认参数即可。为 True 时，需要 param_dict 中包含定点偏置和小数点位置。
- **hw_aligned** – 布尔类型，默认为 True。在运行网络时，是否采用“与硬件一致”的方式；目前仅会影响均值池化（硬件限制均值池化的分母与标准的均值池化有所区别，具体请参考 Pooling）。用户在定点训练时使用默认参数即可。
- **bitwidth** – 定点数位宽，整数类型，取值限定为 8 或者 16，默认为 8。
- **for_training** – 用来标识用户是否将该对象用于定点训练。为 True 时，函数会从 param_dict 中取浮点权重和偏置进行参数的初始化，并且在 quant_output 为 True 时会将偏置增加一个额外值以将该层输出张量补偿为四舍五入，定点训练时使用该方式更容易得到更好地结果；为 False 时，函数会直接读取 param_dict 中定点权重和偏置进行参数的初始化，当用户需要用该对象和传入的 param_dict 模拟芯片定点推理结果，请将该参数设置为 False。默认为 True。

该类有 get_param_dict 函数，用来将网络参数保存成 NDK 约定的 param_dict 形式。

注意事项：

- 该 pytorch 自定义模型的输出结果永远为 4 维张量。例如 imageNet 分类问题对应的网络，输出是 N*C*1*1 的 4 维张量，因为需要使用 reshape 函数将输出变成 N*C 的 2 维张量才能计算损失函数。

- 使用 double 函数将该模型变成双精度类型方可准确无误地模拟芯片行为。
- 网络不会对输入做量化，因此在定点训练时，请将网络输入按照 param_dict 中保存的输入张量小数点位置量化网络输入。一个例外是，如果输入数据是原始 0~255 范围的 8bit 位图，则定点训练 8bit 网络时不需要进行量化。
- 构建优化器的方式和经典 pytorch 训练有所区别，具体请见 pytorch 定点训练示例。

使用 pytorch 定点训练的示例请见 ndk/examples/example_pytorch_nets.py。浮点模型源于 torchvision 提供模型库，并使用 torch.onnx.export 函数保存成 onnx 文件，输入图像为 3 通道 224*224。该网络对图像的预处理流程是先除以 255 得到[0, 1]范围的数据，再减去均值[0.485, 0.456, 0.406]，最后除以标准差[0.229, 0.224, 0.225]。化简该预处理流程后，得到的流程是将原始图像（0~255 范围）乘以[1/0.229/255, 1/0.224/255, 1/0.225/255]，再加上[-0.485/0.229, -0.456/0.224, -0.406/0.225]。因此我们使用预处理工具将预处理部分并入神经网络，weight 参数取 np.array([1/0.229/255, 1/0.224/255, 1/0.225/255])，bias 参数取 np.array([-0.485/0.229, -0.456/0.224, -0.406/0.225])。

4.4 模型打包工具

ndk.modelpack.modelpack

通过模型打包工具，用户能够将经过定点化的 Layer 对象列表和参数字典打包输出为二进制文件（.bin），以供下载到芯片平台。

函数调用形式如下：

```
modelpack( bitwidth,
           layer_list,
           param_dict,
           out_file_path,
           model_name='model'
           )
```

必要参数：

- **bitwidth** – 定点数位宽，整数类型，取值限定为 8 或者 16；
- **layer_list** – Layer 对象的 Python 列表类型，待保存的神经网络模型结构信息；
- **param_dict** – Python 字典类型，待保存的网络模型对应的参数字典；
- **out_file_path** – 字符串对象，指定模型打包工具输出文件保存的目录。

可选参数：

- **model_name** – 字符串对象，模型的名称，用于在 SDK 中加载；如有多个模型需要同时下载到芯片，请分别为其取不同的名称。

返回参数：

- 无

ndk.modelpack.modelpack_from_file

除去以上函数，用户也可以选择将经过定点化、并使用 `save_to_file` 函数保存的模型文件（成对的.prototxt 和.npz 文件）打包输出为二进制文件（.bin），以供下载到芯片平台。

函数调用形式如下：

```
modelpack_from_file( bitwidth,
                     fname_prototxt,
                     fname_npz,
                     out_file_path,
                     model_name='model'
                     )
```

必要参数：

- **bitwidth** – 定点数位宽，整数类型，取值限定为 8 或者 16；
- **fname_prototxt** – 字符串对象，Layer 对象列表导出得到的 prototxt 文件；
- **fname_npz** – 字符串对象，经定点化操作的参数字典所保存的 npz 文件；
- **out_file_path** – 字符串对象，指定模型打包工具输出文件保存的目录；

可选参数：

- **model_name** – 字符串对象，模型的名称，用于在 SDK 中加载；如有多个模型需要同时下载到芯片，请分别为其取不同的名称。

返回参数：

- 无

成功调用该函数后，在 `out_file_path` 指定的输出文件路径下，能够找到一个 `model_name` 指定名称、后缀为.bin 文件，即是模型打包后的二进制文件。

除了这个.bin 文件以外，在输出文件路径下还会生成一些调试用以及高级开发用的文件。

4.5 误差分析工具

ndk.quantize.analyze_tensor_distribution

NDK 提供了给定模型结构和参数，对指定张量的分布进行统计的函数。

对模型中指定张量的分布进行统计的函数调用形式如下：

```
stats_dict = analyze_tensor_distribution(
                                    layer_list,
                                    param_dict,
                                    target_tensor_list,
                                    output_dirname=None,
```

```
                                    data_generator=None,
                                    quant=True,
                                    bitwidth=8,
                                    num_batch=1000,
                                    hw_aligned=True,
                                    num_bin=0,
                                    max_abs_val=None,
                                    log_on=False
                                    )
```

必要参数:

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息;
- **param_dict**– Python 字典类型，网络模型对应的参数字典;
- **target_tensor_list** – Python 列表类型，指定了需要进行统计的张量名，可以为特征张量名或某一层的权重/偏置名（"层名称_weight"/"层名称_bias"）;

可选参数:

- **output_dirname** – 字符串类型，默认 None。

  配置成文件夹路径之后，会在目标文件夹下将返回参数保存成.csv 文件：其中概率分布保存在 prob.csv，其余统计参数保存在 stats.csv;

  注意，功能需要依赖 pandas 模块。如无法引入该模块，请使用 pip install pandas 命令为 python 环境下载安装 pandas 模块。

- **data_generator** – Python 生成器类型，数据样本生成器函数，每次调用 `next()` 时返回字典数据中必须包含一组神经网络模型的输入样本（即'input'键），可以不用给出模型的输出样本（即'output'键）;

  如前所述，输入和输出样本均要求是一个 4 维 numpy.ndarray 对象，按照 NCHW 排列，因此训练的 batch_size 由该对象的第一维大小决定。

  默认值为 None，但此时如果 `target_tensor_list` 包含特征张量名则会报错。

- **quant** – 布尔类型，若为 True 则会读取定点参数进行数据分布统计；若为 False，则读取浮点参数进行数据分布统计;
- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16，默认值为 8;
- **num_batch** – 整数类型，统计特征张量的分布时，根据 `num_batch` 个 batch 的数据进行统计，默认值为 1000;
- **hw_aligned** – 布尔类型，默认为 True。在运行网络时，是否采用"与硬件一致"的方式；目前仅会影响均值池化（硬件限制均值池化的分母与标准的均值池化有所区别，具体请参考《层类型说明》中的 Pooling 小节）;
- **num_bin** – 整数类型，统计分布时所取的 bin 的数量；默认值为 0，此时不对样本分布做记录，只统计最大值、最小值、均值、方差;

- **max_abs_val** – 整数类型，默认值为 None：

  若配置为非 None 的整数，在统计特征张量的分布时，直接根据 `max_abs_val` 确定上下限值；

  若配置为 None，则会先跑足够多的 Batch 直到累计样本数达到 1000 或以上，然后根据结果的绝对值最大值确定 bin 的上下限值。

- **log_on** – 布尔类型，默认值为 False：

  设置为 True 时，打开屏显执行日志信息；

  设置为 False 时，关闭屏显执行日志信息。

返回参数：

- **stats_dict** – Python 字典类型，指定张量的分布统计值，其中键名和值类型作如下约定：

  1) 对于某一层的权重，张量名格式为“层名称_weight”；对于某一层的偏置，张量名格式为“层名称_bias”；对于特征张量，张量名即为该特征张量对应的张量名；

  2) 最小值，键格式为“张量名_min”，值为一个浮点数；

  3) 最大值，键格式为“张量名_max”，值为一个浮点数；

  4) 平均值，键格式为“张量名_avg”，值为一个浮点数；

  5) 方差，键格式为“张量名_var”，值为一个浮点数；

  6) 概率分布，键格式为“张量名_pd”，值是一个大小为 $2\times$`num_bin` 的 numpy.ndarray 对象，第一维记录了各个 bin 对应的中心值，第二维记录了统计过程中对应张量中元素数值落到各个 bin 中的次数；若 `num_bin=0`，则 stats_dict 中不含有此项。

ndk.quantize.compare_float_and_quant

为了方便对定点化前后模型差异进行比较，NDK 提供了给定模型结构和参数，对指定张量在定点后前后的差异进行比较分析的函数。

函数调用形式如下：

```
Result_dict = compare_float_and_quant(
                              layer_list,
                              param_dict,
                              target_tensor_list,
                              bitwidth,
                              output_dirname=None,
                              data_generator=None,
                              num_batch=1000,
                              hw_aligned=True,
                              num_bin=0,
                              max_abs_val=None,
```

```
                                    log_on=False)
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息；
- **param_dict**– Python 字典类型，网络模型对应的参数字典；
- **target_tensor_list** – Python 列表类型，指定了需要进行统计的张量名，可以为特征张量名或某一层的权重/偏置名（”层名称_weight”/“层名称_bias”）；
- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16。

可选参数：

- **output_dirname** – 字符串类型，默认 None。

  配置成文件夹路径之后，会在目标文件夹下将返回参数保存为一个 comp_results.csv 文件；

  注意，功能需要依赖 pandas 模块。如无法引入该模块，请使用 pip install pandas 命令为 python 环境下载安装 pandas 模块。

- **data_generator** – Python 生成器类型，数据样本生成器函数，每次调用 `next()` 时返回字典数据中必须包含一组神经网络模型的输入样本（即’input’键），可以不用给出模型的输出样本（即‘output’键）；

  如前所述，输入和输出样本均要求是一个 4 维 numpy.ndarray 对象，按照 NCHW 排列，因此训练的 batch_size 由该对象的第一维大小决定。

  默认值为 None，但此时如果 `target_tensor_list` 包含特征张量名则会报错。

- **num_batch** – 整数类型，统计特征张量的分布时，根据 `num_batch` 个 batch 的数据进行统计，默认值为 1000；

- **hw_aligned** – 布尔类型，默认为 True。在运行网络时，是否采用“与硬件一致”的方式；目前仅会影响均值池化（硬件限制均值池化的分母与标准的均值池化有所区别，具体请参考 Pooling）

- **num_bin** – 整数类型，统计分布时所取的 bin 的数量；默认值为 0，此时不对样本分布做记录，因此输出数据中不会包含 KL 散度、JS 散度等需要统计数据分布的指标；

- **max_abs_val** – 整数类型，默认值为 None：

  若配置为非 None 的整数，在统计特征张量的分布时，直接根据 `max_abs_val` 确定上下限值；

  若配置为 None，则会先跑足够多的 Batch 直到累计样本数达到 1000 或以上，然后根据结果的绝对值最大值确定 Bin 的上下限值。

- **log_on** – 布尔类型，默认值为 False：

  设置为 True 时，打开屏显执行日志信息；

设置为 False 时，关闭屏显执行日志信息。

返回参数：

- **result_dict** – Python 字典类型，指定张量在浮点和定点模型时的比较值，其中键名和值类型作如下约定：

  1) 对于某一层的权重，张量名格式为“层名称_weight”；对于某一层的偏置，张量名格式为“层名称_bias”；对于特征张量，张量名即为该特征张量对应的张量名；

  2) 浮点均方值，键格式为“张量名_fmsv”，值为一个浮点数；

  3) 定点均方值，键格式为“张量名_qmsv”，值为一个浮点数；

  4) 平均绝对误差，键格式为“张量名_mae”，值为一个浮点数；

  5) 最大绝对误差，键格式为“张量名_maxae”，值为一个浮点数；

  6) 均方误差，键格式为“张量名_mse”，值为一个浮点数；

  7) 量化信噪比，键格式为“张量名_qsnr”，值为一个浮点数；

  8) 最大值序号不一致比率，键格式为“张量名_top1err”，值为一个浮点数；

  9) Kullback-Leibler 散度，键格式为“张量名_kld”，值为一个浮点数，仅当配置 `num_bin` 大于 0 时，输出该指标；

  10) Jensen-Shannon 散度，键格式为“张量名_jsd”，值为一个浮点数，仅当配置 `num_bin` 大于 0 时，输出该指标；

  11) 用于统计数据分布的样本数量，键格式为“张量名_nsamp”，值为整数，仅当配置 `num_bin` 大于 0 时，输出该指标；样本数量相比于输入参数 `num_bin` 过于小时，KL 散度以及 JS 散度的值偏差可能会比较大。

4.6 其它工具函数

4.6.1 保存模型到文件

ndk.modelpack.save_to_file

NDK 通过使用成对的一个 Layer 列表和一个参数字典表示神经网络模型。

在解析完成、优化、定点化之后，随时都可以将当前的网络模型保存到文件，其中 Layer 列表会保存到.prototxt 文件中、参数字典会保存在.npz 文件中。

模型保存函数调用形式如下：

```
save_to_file(layer_list,
             fname_prototxt,
             param_dict=None,
             fname_npz=None
             )
```

必要参数:

- **layer_list** – Layer 对象的 Python 列表类型，待保存的神经网络模型结构信息;
- **fname_prototxt** – 字符串对象，Layer 对象列表导出得到的 prototxt 文件。

可选参数:

- **param_dict** – Python 字典类型，待保存的神经网络模型对应的参数字典;

  默认值为 None，此时仅保存 layer_list 到 prototxt 文件。

- **fname_npz** – 字符串对象，经定点化操作的参数字典所保存的 npz 文件。

  默认值为 None，若此时 `param_dict` 不为 None（即出给了 `param_dict` 但没有指定对应的 npz 文件名）则程序报错。

返回参数:

- 无

4.6.2 从文件加载模型

ndk.modelpack.load_from_file

NDK 通过使用成对的一个 Layer 列表和一个参数字典表示神经网络模型。

前一小节介绍了用户可以通过 `save_to_file()` 函数保存的模型文件，其中 Layer 列表会保存到.prototxt 文件中、参数字典会保存在.npz 文件中。从已经保存的.prototxt 和.npz 文件中加载模型，则需要通过调用函数 `load_from_file()` 实现。

加载模型的函数调用形式如下:

```
layer_list, param_dict = load_from_file(fname_prototxt, fname_npz)
```

必要参数:

- **fname_prototxt** – 字符串对象，Layer 对象列表导出得到的 prototxt 文件;
- **fname_npz** – 字符串对象，经定点化操作的参数字典所保存的 npz 文件。

可选参数:

无

返回参数:

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息;
- **param_dict**– Python 字典类型，网络模型对应的参数字典。

4.6.3 获取层名

ndk.layers.get_layer_name_list

在调用误差分析工具、运行模型观察输出时，用户需要根据模型中各层的名字（即 name 属性）指定目标层的权重或偏置。

用户有两种途径获得所需的层名称：

1) 调用 `save_to_file()` 保存模型，然后从.prototxt 文件查找目标层名；

2) 调用函数 `get_layer_name_list()` 来获取一个包含所有层名的列表。

获取层名列表的函数调用形式如下：

```
layer_name_list = get_layer_name_list(layer_list)
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息。

可选参数：

无

返回参数

- **layer_name_list** – Python 列表类型，层名称的列表。

4.6.4 获取张量名

在调用误差分析工具、运行模型观察输出时，用户会需要指定指定特征张量的名字：

- 对于特征张量，张量名就是该张量的名字（出现在模型某一层的 top 中）；
- 对于权重/偏置，张量名格式为“层名称_weight”/“层名称_bias”。

用户有四种途径获得所需的张量名：

1) 调用 `save_to_file()` 保存模型，然后从.prototxt 文件查找目标张量名/层名；

2) 调用函数 `get_tensor_name_list ()` 获取一个包含所有张量名的列表；

3) 调用函数 `get_network_output ()` 快速获取模型最终的输出张量名的列表；

4) 调用函数 `get_net_input_output ()` 快速获取模型输入和最终的输出张量名的列表。

ndk.layers.get_tensor_name_list

获取张量名列表的函数调用形式如下：

```
tensor_name_list = get_tensor_name_list(
                        layer_list,
                        feature_only=False
```

```
                                )
```

必要参数:

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息。

可选参数:

- **feature_only** – 布尔类型，默认值为 False:

  当设定为 True 时，返回列表中仅包含特征张量名;

  当设定为 False 时，返回列表中除特征张量名外，还包含各层的权重/偏置。

返回参数:

- **tensor_name_list** – Python 列表类型，张量名的列表。

ndk.layers.get_network_output

获取模型输出张量名的函数调用形式如下:

```
output_tensor_name_list = get_network_output(layer_list)
```

必要参数:

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息。

可选参数:

无

返回参数:

- **output_tensor_name_list**– Python 列表类型，模型输出张量名的列表（通常情况下模型输出张量唯一，此时仅包含一个元素，）。

ndk.layers.get_net_input_output

获取模型输入和输出张量名的函数调用形式如下:

```
input_tensor_name_list, output_tensor_name_list =
get_net_input_output(layer_list)
```

必要参数:

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息。

可选参数:

无

返回参数:

- **input_tensor_name_list**– Python 列表类型，模型输入张量名的列表（通常情况下模型输入张量唯一，此时仅包含一个元素，）。

- **output_tensor_name_list**– Python 列表类型，模型输出张量名的列表（通常情况下模型输出张量唯一，此时仅包含一个元素，）。

4.6.5 获取神经网络的权重/偏置值

ndk.quant_tools.numpy_net.get_layer_weight_bias

通过调用 `get_layer_weight_bias()` 函数能够获取参数字典中指定层权重或偏置的值。若要求查看定点数的权重和偏置，则会根据参数字典中对应定点整数和小数点位置计算该定点数，并返回该定点数的浮点表示。

函数调用形式如下：

```
data = get_layer_weight_bias(target_tensor_name,
                             param_dict,
                             quant=False,
                             bitwidth=8
                             )
```

必要参数：

- **target_tensor_list** – Python 列表类型，指定了需要进行统计的张量名，可以为特征张量名或某一层的权重/偏置名（"层名称_weight"/"层名称_bias"）；
- **param_dict** – Python 字典类型，网络模型对应的参数字典；

可选参数：

- **quant** – 布尔类型，默认为 False：

  若为 True 则读取定点权重/偏置；

  若为 False 则读取浮点权重/偏置。

- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16，默认为 8。

返回参数：

- **data** – numpy.ndarray 类型的对象，目标权重/偏置的值

4.6.6 运行神经网络并获取特征张量的值

ndk.quant_tools.numpy_net.run_layers

如果在下载到芯片平台之前，还需要对模型运行数据进行分析（如统计识别率）等，NDK 提供了可以在 PC 上运行的函数。

函数调用形式如下：

```
data_dict = run_layers(input_data_batch,
                       layer_list,
```

```
                        target_feature_tensor_list,
                        param_dict=None,
                        quant=False,
                        bitwidth=8,
                        hw_aligned=False,
                        numpy_layers=None,
                        log_on=False,
                        use_ai_framework=True,
                        quant_weight_only=False,
                        quant_feature_only=False
                        )
```

必要参数:

- **input_data_batch** – numpy.ndarray 类型，一个 batch 的网络输入数据，必须是 4 维，NCHW 格式。
- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息；
- **target_feature_tensor_list** – Python 列表类型，该列表指定了需要进行统计的张量名，仅允许包含特征张量名；

可选参数:

- **param_dict**– Python 字典类型，网络模型对应的参数字典，默认值为 None；

  配置了 numpy_layers 参数时，函数将不再读取 `param_dict`，此时 `param_dict` 可以不进行配置。
- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16，默认值为 8；
- **quant** – 布尔类型，是否使用定点模型参数，默认值为 True；

  若为 True 则会读取定点参数进行模型计算；

  若为 False 则读取浮点参数进行模型计算；
- **hw_aligned** – 布尔类型，默认为 False。在运行网络时，是否采用“与硬件一致”的方式；目前仅会影响均值池化（硬件限制均值池化的分母与标准的均值池化有所区别，具体请参考 Pooling）
- **numpy_layers** – Python 列表，由 ndk.quant_tools.quant_layers.QuantLayer 类型对象组成，可以利用函数 `build_numpy_layers()` 预先创建。

  配置了 `numpy_layers` 参数时，函数将会跳过定点层对象的创建阶段，直接进行数值计算。在需要跑多个 batch 的数据而反复调用本函数时，这样做可以有效提高运行效率。此时，本函数不再读取 `param_dict`，因此 `param_dict` 可以不进行配置。
- **log_on** – 布尔类型，默认值为 False：

  设置为 True 时，打开屏显执行日志信息；

  设置为 False 时，关闭屏显执行日志信息。

- **use_ai_framework** – 布尔类型，默认值为 True：

  设置为 True 时，底层创建 TensorFlow/Pytorch 计算图以加速卷积、池化等运算；

  设置为 False 时，仅使用 numpy 运算。

- **quant_weight_only** – 布尔类型，默认值为 False，仅当 `quant=True` 时有效：

  设置为 True 并且 `quant=True` 时，在执行网络层时，仅对权重和偏置使用定点数，特征张量依然使用浮点数；

  当 `quant=True` 时，本参数与 `quant_feature_only` 不允许同时设置为 True。

- **quant_feature_only** – 布尔类型，默认值为 False，仅当 `quant=True` 时有效：

  设置为 True 并且 `quant=True` 时，在执行网络层时，仅对特征张量使用定点数，权重和偏置依然使用浮点数；

  当 `quant=True` 时，本参数与 `quant_weight_only` 不允许同时设置为 True。

返回参数：

- **data_dict** – Python 字典类型，其中，键名为指定特征张量的名称、值为该张量对应的神经网络计算结果。

ndk.quant_tools.numpy_net.build_numpy_layers

在需要多次调用 `run_layers()` 函数跑多个 batch 时，为避免反复地创建对象、提高执行效率，建议使用 `build_numpy_layers()` 函数预先创建 QuantLayer 类型的对象列表。

调用形式如下：

```
numpy_layers= build_numpy_layers(layer_list,
                                 param_dict,
                                 quant=False,
                                 bitwidth=8
                                 )
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息；
- **param_dict**– Python 字典类型，网络模型对应的参数字典。

可选参数：

- **quant** – 布尔类型，是否使用定点模型参数，默认值为 True；

  若为 True 则会根据定点参数创建 QuantLayers 对象；

  若为 False 则会根据浮点参数创建 QuantLayers 对象；

- **bitwidth** – 整数类型，定点数位宽，可选配为 8 或者 16，默认值为 8；

返回参数：

- **numpy_layers** – Python 列表，由 ndk.quant_tools.quant_layers.QuantLayer 类型对象组成。

4.6.7 检查神经网络是否被硬件支持

ndk.layers.check_layers

用户可以从 Caffe 框架或者 TensorFlow 框架下的模型导入到 NDK，也可以手工定义调用 `ndk.layers.Layer` 里面的类来创建模型；在定点化的时候，可以通过调用 `ndk.quantize` 下的工具函数，也可以手动修改 `param_dict` 里的参数。定义网络结构或者调整参数时必须要满足《层类型说明》中列出的对种类型网络层的限制。

网络结构和参数检查，可以通过调用函数 `check_layers` 实现。

调用形式如下：

```
flag_warning = check_layers(layer_list, param_dict=None)
```

必要参数：

- **layer_list** – Layer 对象的 Python 列表类型，网络模型结构信息；

可选参数：

- **param_dict**– Python 字典类型，网络模型对应的参数字典。

返回参数：

- **flag_warning** – 布尔类型，是否存在风险。若存在风险返回值为 True，函数会将警告信息打印在命令行中。

注意：目前版本的 `check_layers` 函数仅能对网络结构模型和参数字典中浮点参数进行检查，暂不支持对定点参数的检查。

4.6.8 比对 NDK 和芯片运行神经网络结果

NDK 支持将 run_layers 函数（具体请见 4.6.6 节）的结果以数组形式保存成 C 语言头文件。用户可以将该头文件的数据给 NPU 运行神经网络。

ndk.utils.save_quantized_feature_to_header

调用形式如下：

```
save_quantized_feature_to_header(filename,
                                 feature,
                                 bit_width,
                                 frac,
                                 signed = True,
                                 name_in_header = 'feature',
                                 aligned = True)
```

必要参数：

- **filename** – 头文件路径；
- **feature** – 4D numpy.ndarray 类型，待保存的张量；
- **bit_width** – 整数类型，定点数位宽，可选配为 8 或者 16；
- **frac** – 该张量的定点小数点位置。

可选参数：

- **signed** – 布尔类型，该张量是否为有符号数，默认值为 True；
- **name_in_header** – 字符串类型，头文件中数组名称，默认为"feature"；
- **aligned** – 布尔类型，该张量是否按照每行 32 字节对齐编排，默认值为 True；
  - **SDK 串行调用方式（cnn_run_net_pack 函数）需要该参数为 False，其他模式需要该参数为 True。**

一种典型的 run_layers 和保存头文件的代码组合方式如下，这里 quant_layer_list 和 quant_param_dict 保存了定点网络的信息：

```
in_names, out_names = get_net_input_output(quant_layer_list)
in_name = in_names[0]
out_name = out_names[0]

data_dict = run_layers(input_data_batch,
                       quant_layer_list,
                       in_names + out_names,
                       quant_param_dict,
                       quant=True
                       bitwidth=my_own_bitwidth,
                       hw_aligned=True,
                       numpy_layers=None,
                       log_on=False,
                       use_ai_framework=True,
                       quant_weight_only=False,
                       quant_feature_only=False
                       )

in_frac = quant_param_dict[in_name + "_frac"]
out_frac = quant_param_dict[out_name + "_frac"]
in_signed = quant_param_dict[in_name + "_signed"]
out_signed = quant_param_dict[out_name + "_signed"]

save_quantized_feature_to_header('input.h',
                                 data_dict[in_name],
                                 my_own_bitwidth,
                                 in_frac,
```

```
                                    signed = in_signed,
                                    name_in_header = 'in_feature'
                                    )
save_quantized_feature_to_header('output.h',
                                    data_dict[out_name],
                                    my_own_bitwidth,
                                    out_frac,
                                    signed = out_signed,
                                    name_in_header = 'out_feature'
                                    )
```

然后将得到的 input.h 里面的数据作为芯片上网络的输入，并且比对 output.h 的数据和芯片上得到的网络输出。

五、 SDK 调用接口

通过模型打包工具将模型打包为二进制文件之后（参见《模型打包工具》），就可以在物奇 SDK 中创建工程调用 NPU 进行计算了。详细的调用函数接口声明、内存排放规则和其它使用技巧，请参考 SDK 相关文档，本文档仅对此作简要介绍。

在进行代码编写之前，首先将模型打包后得到 bin 文件拷贝到工程目录下的 custom 路径下（注意请将该路径下所有无关的文件都删除）。

SDK 中 NPU 调用有两种方式。

- 串行方式：NPU 在进行计算时，CPU 等待，直到 NPU 返回结果后，CPU 继续执行下一条指令。
- 并行方式：NPU 在进行计算时，允许 CPU 同时做其它运算，例如从 flash 加载另一个模型、做前一张图片结果的后处理运算等。

5.1 串行调用方式

串行调用方式使用简单，适用于模型的快速验证，或者程序的全部后续工作都需要依赖 NPU 计算结果的情况。在 NPU 在进行计算时，CPU 等待；直到 NPU 返回结果后，CPU 才会继续执行下一条指令。

第一步：根据名字从 flash 中加载神经网络模型，

```
struct cnn_net_pack_info cnn_load_net_pack(char* net_name);
```

作为输入参数的网络名称为 bin 文件的名称，如“abc.bin”；函数返回一个网络参数结构体，该结构体定义如下：

```
struct cnn_net_pack_info {
    uint8_t deleted;      /* whether this pack is deleted */
    uint32_t input_addr;  /* address of input image of this pack */
    uint32_t output_addr; /* address of output image of this pack*/
    void     *pack;           /* pointer of the pack */
    uint16_t input_channel;    /* input channel of the net */
    uint16_t input_height;     /* input height of the net */
```

```
    uint16_t input_width;       /* input width of the net */
    uint16_t output_channel;    /* output channel of the net */
    uint16_t output_height;     /* output height of the net */
    uint16_t output_width;      /* output width of the net */
    uint8_t feature_byte;       /* 1: 8-bit, 2: 16-bit */
};
```

第二步：准备输入数据，注意必须是 NCHW 格式，其中 N 为 1；然后分配一块内存用以存放网络的输出结果，可以用以下函数分配内存：

```
void* os_mem_malloc( uint16_t module_id,
                              uint32_t bytes);
```

第三步：执行模型计算，三个参数分别为指向网络结构体的指针，输入图像指针和输出指针。结果会保存到指定的输出地址。

```
uint8_t cnn_run_net_pack(struct cnn_net_pack_info *pack_info,
                                                  void *input,
                                                  void *output);
```

第四步：请及时释放网络参数结构体占用的内存。

```
void cnn_delete_net_pack(struct cnn_net_pack_info *pack_info);
```

5.2 并行调用方式

并行方式调用相对复杂一些，要求用户对物奇 NPU 数据存储方法、对齐等有一定的了解。并行调用方式能够有效提高 CPU 和 NPU 的利用率，适用于对资源利用率要求较高的开发场景。

第一步：调用 `cnn_load_net_pack` 函数，根据名字从 flash 中加载神经网络模型，函数返回一个 `cnn_net_pack_info` 类型的网络参数结构体 `pack_info`；

第二步：确定输入数据已存放在 `pack_info.input_addr` 指向的地址，请不要直接更改该指针。直接将输入数据写到该地址，或者通过以下函数将数据搬移至该地址，

```
void os_mem_cpy(void *dst, const void *src, uint32_t len);
```

第三步：启动模型计算。以下函数仅仅是启动 NPU 的计算过程，启动之后 CPU 可以执行其它指令。

```
uint8_t cnn_start_net_pack(struct cnn_net_pack_info *pack_info);
```

该函数返回值含义为：

- 0：成功启动 NPU 计算；
- 1：该 pack_info 指定的模型已从内存被删除；
- 2：NPU 正忙，无法启动当前计算任务；
- 3：定点位宽（feature_byte）配置错误

第四步：监控 NPU 计算状态，在计算结束时读取结果，计算结果需要用户从网络参数结构体 `pack_info` 中通过 `pack_info.output_addr` 地址指针读取（数据内容按 32 字节对齐）。
监控 NPU 计算状态的函数为：

```
int8_t cnn_get_status();
```

返回值含义为：

- 0：NPU 空闲（已完成之前指派的计算任务）
- 非零值：NPU 正忙

第五步：请及时调用 `cnn_delete_net_pack` 函数，释放网络参数结构体占用的内存。

六、 层类型说明

NDK 中所有类型层的输入、输出特征均是 NCHW 格式的 4 维张量，四个维度含义依次分别是 Batch 大小、通道数、高度、宽度。例如：输入为 K 维输出为 N 维的全连接层，那么输入张量各维度大小为(:, K, 1, 1)、输出张量各维度大小为(:, N, 1, 1)。

6.1 Input

Input 层指定了整个网络的输入维度。作为整个网络的输入，Input 层只有 top 没有 bottom。

Input 层有两个个特殊属性：

- dim – 整数类型四元组，分别表示网络输入（Input 层的输出，top）张量的四个维度。其中，第一维参数在 NDK 中没有作用，设置成 1 即可；之后三维依次为：通道数、高、宽。

prototxt 文件中的层定义示例：

```
layer {
  name: "input"
  type: "Input"
  top: "net_input"
  input_param {
    shape {
     dim: 1 # useless in NDK
     dim: 3 # number of channels
     dim: 32 # height
     dim: 32 # width
    }
  }
}
```

定点化后，参数字典中存有 Input 层的输出张量（也就是网络输入张量）的量化参数。

例如，输出张量名为"net_input"的 Input 层经过定点化后，参数字典中应有对应的 net_input_frac 和 net_input_signed 字段。

由于硬件的限制，

- 在 8 比特定点化时，Input 层对应的输出张量(top)允许设定"张量名_signed"的值为 False，即整个网络的输入允许做无符号定点化处理，其它类型层（除 Slice、Concat、ShuffleChannel 这些内容搬移的操作外）的输出张量只允许作有符号定点化；
- 在 16 比特定点化时，所有类型层（括 Input 层）的输出只允许做有符号定点化，即所有"张量名_signed"键或者不定义（如果没有这个键，默认为有符号数），或者对应的值必须为 True。

6.2 Simoid

给定输入x，Sigmoid 层的输出为

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Sigmoid 层无特有属性。

prototxt 文件中的层定义示例：

```
layer {
  name: "sig1"
  type: "Sigmoid"
  bottom: "tensor_in"
  top: "tensor_out"
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 Sigmoid 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，定点 Sigmoid 层的参数有以下约束：

- 在 8 比特位宽定点化时，Sigmoid 层的输入张量限定为 1 比特符号位、3 比特整数位、4 比特小数位，即 tensor_in_frac=4；输出张量限定为 1 比特符号位、7 比特小数位，即小数点位置 tensor_out_frac=7；
- 在 16 比特位宽定点化时，Sigmoid 层的输入张量限定为 1 比特符号位、3 比特整数位、12 比特小数位，即小数点位置 tensor_in_frac=12；输出张量限定为 1 比特符号位、15 比特小数位，即小数点位置 tensor_out_frac=15；
- 在 bottom/top 中指定的张量不能够出现在其它非线性层的 top/bottom 里，即不支持连续两次或更多次的非线性运算，包括 Sigmoid、ReLU、ReLU6、TanH。

6.3 ReLU

给定一个输入值x，ReLU 层的输出为：`x > 0? x : negative_slope*x`，如未给定参数 negative_slope 的值，则为标准 ReLU 方法：`max(x, 0)`。

$$\mathrm{ReLU}(x) = \begin{cases} \text{negativ\_slope} * x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

ReLU 层有一个特有属性 negative_slope：大于等于 0 的浮点数，默认为 0。

prototxt 文件中的层定义示例：

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "tensor_in"
  top: "tensor_out"
  relu_param {
     negative_slope: 0.015625
  }
}
```

ReLU 层在参数字典中无对应属性。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 ReLU 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，定点 ReLU 层的参数有以下约束：

- ReLU 层输出张量与输入张量的位宽、小数点位置完全相同；
- negative_slope 在定点化时无论选定 16 比特还是 8 比特量化，该值只能被量化为 8 比特，小数点位置 6，即最小非零取值为 1/64=0.015625；
- 在 bottom/top 中指定的张量不能够出现在其它非线性层的 top/bottom 里，即不支持连续两次或更多次的非线性运算，包括 Sigmoid、ReLU、ReLU6、TanH。

6.4 ReLU6

ReLU6 是 ReLU 的一个变种。给定一个输入值x，ReLU6 层的输出为

$$\mathrm{ReLU6}(x) = min(max(x, 0), 6)$$

ReLU6 层无特有属性。

prototxt 文件中的层定义示例：

```
layer {
  name: "relu6"
  type: "ReLU6"
  bottom: "tensor_in"
  top: "tensor_out"
```

```
}
```

ReLU6 层在参数字典中无对应属性。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 ReLU6 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，定点 ReLU6 层的参数有以下约束：

- 在 8 比特位宽定点化时，ReLU6 层的输入和输出张量限定为 1 比特符号位、3 比特整数位、4 比特小数位，即小数点位置 tensor_in_frac=tensor_out_frac=4；
- 在 16 比特位宽定点化时，ReLU6 层的输入和输出张量限定为 1 比特符号位、3 比特整数位、12 比特小数位，即小数点位置 tensor_in_frac=tensor_out_frac=12。
- 在 bottom/top 中指定的张量不能够出现在其它非线性层的 top/bottom 里，即不支持连续两次或更多次的非线性运算，包括 Sigmoid、ReLU、ReLU6、TanH。

6.5 TanH

给定输入x，TanH 层的输出为

$$\mathrm{TanH}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

TanH 层无特有属性。

prototxt 文件中的层定义示例：

```
layer {
  name: "layer"
  type: "TanH"
  bottom: "tensor_in"
  top: "tensor_out"
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 TanH 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，定点 TanhH 层的参数有以下约束：

- 在 8 比特位宽定点化时，TanH 层的输入张量限定为 1 比特符号位、2 比特整数位、5 比特小数位，即 tensor_in_frac=5；输出张量限定为 1 比特符号位、7 比特小数位，即 tensor_out_frac=7；
- 在 16 比特位宽定点化时，TanH 层的输入张量固定为 1 比特符号位、2 比特整数位、13 比特小数位，即 tensor_in_frac=13；输出张量固定为 1 比特符号位、15 比特小数位，即 tensor_out_frac=15；；
- 在 bottom/top 中指定的张量不能够出现在其它非线性层的 top/bottom 里，即不支持连续两次或更多次的非线性运算，包括 Sigmoid、ReLU、ReLU6、TanH。

6.6 InnerProduct

全连接层。输入向量大小为K，输出向量大小为N，权重为$N \times K$矩阵（实际按 4 维存储），偏置为N维向量。

值得注意的是，InnerProduct 输入是一个四维张量，若输入张量高度 H 和宽度 W 的值不为 1，比如输入张量大小为(:, C, H, W)，等效的输入向量维度$K = C \times H \times W$，此时输出张量的大小仍然是(:, N, 1, 1)。

InnerProduct 层有两个特有属性:

- num_output - 整数类型对象，输出 tensor 的大小;
- bias_term - 布尔类型对象，是否使用偏置，默认为 true。

prototxt 文件中的层定义示例:

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "tensor_in"
  top: "tensor_out"
  inner_product_param {
    num_output: 128
    bias_term: true
  }
}
```

浮点模型在参数字典中存有对应的浮点数权重和浮点数偏置值。例如，名为 ip1 的 InnerProduct 层在参数字典中会有对应的 ip1_weight 和 ip1_bias (若 bias_term=true)。其中权重矩阵维度为(N, C, H, W)，即 `ip1_weight.shape=(N,C,H,W)`，其中$C \times H \times W = K$。

定点化后，参数字典中应有对应的量化权重/偏置值 ip1_quant_weight、ip1_quant_bias 字段，以及权重、偏置、输出张量对应的小数点位置 ip1_frac_weight、ip1_frac_bias、tensor_out_frac。

由于硬件限制，InnerProduct 层的参数有以下约束:

- 输入张量高度和宽度均不能超过 16;
- 定点化参数中，若 bias_term=True，输入特征张量、权重、偏置需要满足:
  - 8 比特定点化时, 0 ≤ tensor_in_frac + ip1_frac_weight - ip1_frac_bias ≤ 15;
  - 16 比特定点化时, 0 ≤ tensor_in_frac + ip1_frac_weight - ip1_frac_bias ≤ 31;
- 定点化参数中，输入特征张量、权重、输出特征需要满足:
  - 8 比特定点化时, 0 ≤ tensor_in_frac+ ip1_frac_weight - tensor_out_frac ≤ 15;
  - 16 比特定点化时, 0 ≤ tensor_in_frac+ip1_frac_weight - tensor_out_frac ≤ 31。

6.7 Convolution

卷积层。输入通道数C_{in}，输出通道数C_{out}，输入特征图高度H、宽度W，输入和输出的特征图按 NCHW 格式。权重为$C_{out} \times (C_{in}/g) \times h \times w$矩阵，偏置为$C_{out}$维向量，其中$h$、$w$分别是卷积核在高度和宽度方向上的大小，$g$为分组数。

Convolution 层具有以下特有属性:

- num_output - 整型，输出通道数，即C_{out};
- kernel_size – 整型二元组，(kernel_h, kernel_w)，高/宽两个方向的卷积核的大小，即(h, w);
- stride – 整型二元组，(stride_h, stride_w)，高/宽两个方向的卷积核的步长;
- pad – 整型四元组，(pad_n, pad_s, pad_w, pad_e)，上/下/左/右四个方向的边缘填充，只支持填零；要求(H+pad_n+pad_s-h)正好能够整除 stride_h，并且(W+pad_n+pad_s-w)正好能够整除 stride_w;

  其中，pad_s/pad_e 允许为负值，其含义是：直接丢弃输入特征张量里最下/右沿的若干行/列。
- bias_term - 布尔类型，是否使用偏置，默认为 true;
- dilation – 整型二元组, (dilation _h, dilation _w), 高/宽两个方向的卷积核膨胀系数;
- group – 整型，分组数，即g。

prototxt 文件中的层定义示例:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "tensor_in"
  top: "tensor_out"
  convolution_param {
    num_output: 160
    kernel_size_h: 3
    kernel_size_w: 3
    stride_h: 2
    stride_w: 2
    pad_n: 1
    pad_s: 1
    pad_w: 1
    pad_e: 1
    bias_term: false
    dilation_h: 1
    dilation_w: 1
    group: 2
  }
```

```
}
```

浮点模型在参数字典中存有对应的浮点数权重和浮点数偏置值。例如，名为“conv1”、输出张量名为“tensor_out”的 Convolution 层在参数字典中应有对应的 conv1_weight 和 conv1_bias（若 bias_term=true）。其中，权重矩阵维度为$(C_{out}, C_{in}/g, \mathrm{h}, w)$，即 `conv1_weight.shape=(c_out, c_in/g, h, w)`。

定点化后，参数字典中会有对应的量化权重/偏置值 conv1_quant_weight、conv1_quant_bias字段，以及权重、偏置、输出张量对应的小数点位置conv1_frac_weight、conv1_frac_bias、tensor_out_frac。

由于硬件限制，Convolution 层的参数有以下约束：

- 卷积核高度 kernel_size_h 和宽度 kernel_size_w 均不超过 16。
- 定点化参数中，若 bias_term=True，输入特征张量、权重、偏置需要满足：
  - 8 比特定点化时，0≤tensor_in_frac+conv1_frac_weight-conv1_frac_bias≤15；
  - 16 比特定点化时，0≤tensor_in_frac+conv1_frac_weight-conv1_frac_bias≤31；
- 定点化参数中，输入特征张量、权重、输出特征张量需要满足：
  - 8 比特定点化时，0≤tensor_in_frac+conv1_frac_weight-tensor_out_frac≤15；
  - 16 比特定点化时，0≤tensor_in_frac+conv1_frac_weight- tensor_out_frac≤31；
- 对于分组卷积，即 group>1，每组的输出通道数要求为 16 的整数倍或者恰好为 1；
- 8bit 定点化时，要求 kernel_size_h×kernel_size_w×每组的输入通道数≤8192，比如 3×3 卷积每组输入通道数不能超过 910；
- 16bit 定点化时，要求 kernel_size_h×kernel_size_w×每组的输入通道数≤4096，比如 3×3 卷积每组输入通道数不能超过 455；
- 目前芯片平台仅支持 stride_h 和 stride_w 相同的情况，并且取值不能超过 6；
- 目前芯片平台仅支持 dilation_h 和 dilation_w 相同的情况，取值不能超过 6；
- 空洞卷积的等效视野范围，即 dilation * (kernel_size – 1) + 1 不能超过 32。

6.8 Pooling

池化层。输入特征图高度H、宽度W，输入和输出的特征张量按 NCHW 格式。

Pooling 层具有以下特有属性：

- kernel_size – 整型二元组，(kernel_h, kernel_w)，高/宽两个方向的卷积核的大小；
- stride – 整型二元组，(stride_h, stride_w)，高/宽两个方向的卷积核的步长；
- pad – 整型四元组，(pad_n, pad_s, pad_w, pad_e)，上/下/左/右四个方向的边缘填充，均值池化时填零，最大值池化时填充 0x80（8 比特量化）或者 0x8000（16 比特量化）；要求(H+pad_n+pad_s-h)正好能够整除 stride_h，并且(W+pad_w+pad_e-w)正好能够整除 stride_w；

其中，pad_s/pad_e 允许为负值，其含义是：直接丢弃输入特征张量里最下/右沿的若干行/列。

- dilation – 整型二元组, (dilation _h, dilation _w), 高/宽两个方向的卷积核膨胀系数；
- pool – 字符串对象，池化方法，默认为'MAX'，也可配置为'AVE'。

prototxt 文件中的层定义示例：

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "tensor_in"
  top: "tensor_out"
  pooling_param {
    kernel_size_h: 3
    kernel_size_w: 3
    stride_h: 2
    stride_w: 2
    pad_n: 1
    pad_s: 1
    pad_w: 1
    pad_e: 1
    dilation_h: 1
    dilation_w: 1
    pool: MAX
  }
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 Pooling 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，Pooling 层的参数有以下约束：

- 池化核在高、宽两个方向上各自不允许超过 16，即

$$\text{kernel\_h} \leq 16,\ \text{kernel\_w} \leq 16$$

  如果遇到全局池化操作，在调用 NDK 模型解析的时候，会自动将该操作等效成若干个连续的 Poolling 层；

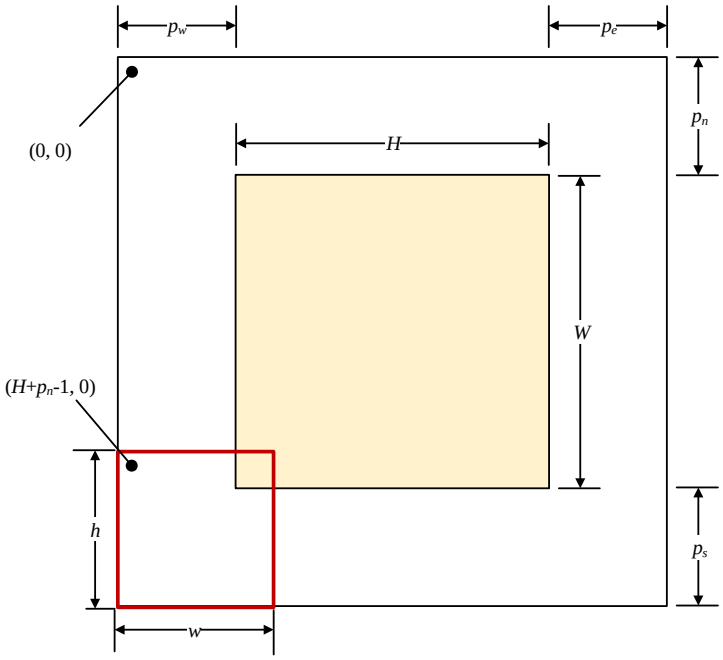
- 配置为最大值池化（'MAX'）时，要求输出张量与输入张量的位宽、小数点位置完全相同；
- 目前芯片平台仅支持 stride_h 和 stride_w 相同的情况，并且取值不能超过 6；
- 目前芯片平台仅支持 dilation_h 和 dilation_w 相同的情况，取值不能超过 6；
- 空洞卷积的等效视野范围，即 dilation * (kernel_size – 1) + 1 不超过 32。

- 配置为均值池化（'AVE'）并且存在边缘填充的情况下，部分输出特征图边缘的值与通常标准意义的均匀池化有所区别，说明如下：

  若当大小为$h \times w$的池化窗在坐标为(i,j)时窗内的元素为$(a_0, a_1, \cdots, a_{n-1})$，均值池化操作对应的输出输出为

  $$s_{i,j} = \frac{\sum_{k=0}^{n-1} a_k}{n_{i,j}}$$

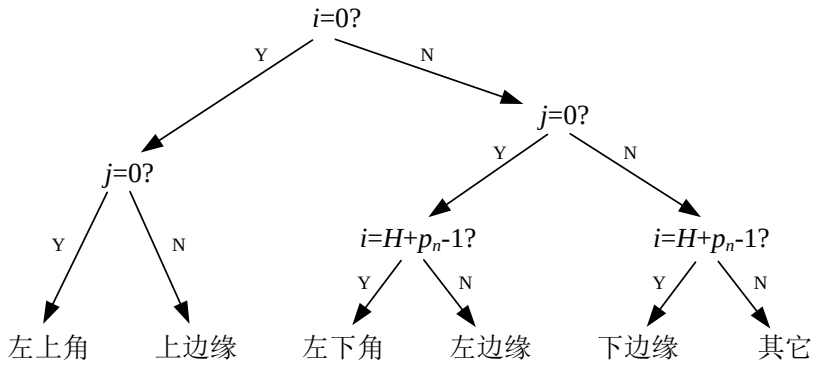
  其中，池化窗的坐标等于窗内左上角元素在填充后特征图中的坐标。



  若无边缘填充即$\text{pad} = (p_n, p_s, p_w, p_e) = (0,0,0,0)$时，除数固定不变，$n_{i,j} = h \times w$。在有边缘填充时，$n_{i,j}$的取值*近似*等于窗内非填充元素的数量。之所以描述为"近似"，是由于硬件对存储的 pattern 数量有限制，Pooling 层只对填充后特征图的左上角、左边缘、左下角、上边缘、下边缘、其它这六类位置作区分：

  - 左上角：$n_{0,0} = (h - p_n) \times (w - p_w)$
  - 左下角：$n_{H+p_n-1,0} = (h - p_s) \times (w - p_w)$
  - 左边缘：$n_{i,0} = h \times (w - p_w)$，$1 \le i \le H + p_n - 2$
  - 上边缘：$n_{0,j} = (h - p_n) \times w$，$j \ge 1$
  - 下边缘：$n_{H+p_n-1,j} = (h - p_s) \times w$，$j \ge 1$
  - 其它：$n_{i,j} = h \times w$，$1 \le i \le H + p_n - 2$且$j \ge 1$

  以上 pattern 各个池化窗位置类别的判决过程可以用下图表示：



6.9 BatchNorm

对于每个通道的输入x，BatchNorm 层的输出为

$$\text{BatchNorm}(x) = wx + b$$

对应于 Caffe 的 BatchNorm 层，$w = \frac{1}{\sigma}$，$b = -\frac{\mu}{\sigma}$，其中μ和σ分别为统计得到的均值和方差。

BatchNorm 层无特有属性。

prototxt 文件中的层定义示例：

```
layer {
    name: "bn1"
    type: "BatchNorm"
    bottom: "tensor_in"
    top: "tensor_out"
}
```

浮点模型在参数字典中存有对应的浮点数权重和浮点数偏置值。例如，名为 bn1 的 BatchNorm 层在参数字典中会有对应的 bn1_weight 和 bn1_bias。

定点化后，参数字典中应有对应的量化权重/偏置值 bn1_quant_weight、bn1_quant_bias 字段，以及权重、偏置、输出张量对应的小数点位置 bn1_frac_weight、bn1_frac_bias、tensor_out_frac。

由于硬件限制，BatchNorm 层的参数有以下限制：

- 归一化操作只能沿着通道方向做，因此 BatchNorm 层对应的权重和偏置值均是C维向量，C为通道数，即每个通道对应一对不同的权重和偏置值。
- 定点化参数中，输入特征张量、权重、偏置以及输出特征张量需要满足：
  - 8 比特定点化时，0 ≤ tensor_in_frac+bn1_frac_weight-bn1_frac_bias ≤ 15；
  - 8 比特定点化时，0 ≤ tensor_in_frac+bn1_frac_weight-tensor_out_frac ≤ 15；
  - 16 比特定点化时，0 ≤ tensor_in_frac+bn1_frac_weight-bn1_frac_bias ≤ 31；
  - 16 比特定点化时，0 ≤ tensor_in_frac+bn1_frac_weight-tensor_out_frac ≤ 31；

6.10 Bias

对于每个通道的输入x，Bias 层的输出为

$$\mathrm{Bias}(x) = x + b$$

Bias 层有无特有属性。

prototxt 文件中的层定义示例：

```
layer {
    name: "bias1"
    type: "Bias"
    bottom: "tensor_in"
    top: "tensor_out"
}
```

浮点模型在参数字典中存有对应的浮点数偏置值。例如，名为 bias1 的 Bias 层在参数字典中会有对应的 bias1_bias。

定点化后，参数字典中应有对应的量化偏置值 bias1_quant_bias 字段，以及偏置、输出张量对应的小数点位置 bias1_frac_bias、tensor_out_frac。

由于硬件限制，Bias 层的参数有以下限制：

- 加偏置操作只能沿着通道方向做，因此 Bias 层对应的偏置值是C维向量，C为通道数，即每个通道对应一个不同的偏置。
- 定点化参数中：
  - 8 比特定点化时，0 ≤ tensor_in_frac - bias1_frac_bias ≤ 15；
  - 8 比特定点化时，0 ≤ tensor_in_frac - tensor_out_frac ≤ 15；
  - 16 比特定点化时，0 ≤ tensor_in_frac - bias1_frac_bias ≤ 31；
  - 16 比特定点化时，0 ≤ tensor_in_frac - tensor_out_frac ≤ 31；

6.11 Scale

Scale 层与 BatchNorm 层的操作几乎相同。对于每个通道的输入x，Scale 层的输出为

$$\mathrm{Scale}(x) = wx + b$$

Scale 层有一个特有属性：

- bias_term - 布尔类型对象，是否使用偏置，默认为 true。

prototxt 文件中的层定义示例：

```
layer {
    name: "scale1"
    type: "Scale"
    bottom: "tensor_in"
```

```
    top: "tensor_out"
    scale_param {
        bias_term: true
    }
}
```

浮点模型在参数字典中存有对应的浮点数权重和浮点数偏置值。例如，名为 scale1 的 Scale 层在参数字典中会有对应的 scale1_weight 和 scale1_bias。

定点化后，参数字典中应有对应的量化权重/偏置值 scale1_quant_weight、scale1_quant_bias 字段，以及权重、偏置、输出张量对应的小数点位置 scale1_frac_weight、scale1_frac_bias、tensor_out_frac。

由于硬件限制，Scale 层的参数有以下限制：

- 归一化操作只能沿着通道方向做，因此 Scale 层对应的权重和偏置值均是C维向量，C为通道数，即每个通道对应一对不同的权重和偏置值。
- 定点化参数中，若 bias_term=True，输入特征张量、权重、偏置需要满足：
  - 8 比特定点化时，0≤tensor_in_frac+ scale1_frac_weight- scale1_frac_bias≤15；
  - 16 比特定点化时，0≤tensor_in_frac+ scale1_frac_weight- scale1_frac_bias≤31；
- 定点化参数中，输入特征张量、权重、输出特征张量需要满足：
  - 8 比特定点化时，0≤tensor_in_frac+ scale1_frac_weight-tensor_out_frac≤15；
  - 16 比特定点化时，0≤tensor_in_frac+ scale1_frac_weight-tensor_out_frac≤31。

6.12 Slice

Slice 层将一个输入 tensor 分解成多个输出 tensor。其对应的 Layer 对象中，bottom 的值为字符串类型的输入张量名，top 的值为输出张量名的字符串列表。输出张量名列表 top 中不允许有重复的张量名。

Slice 层有两个特有属性：

- axis - 整数类型对象，切分维度（目前只支持沿着通道维度切分，即 axis=1）；
- slice_point – 整型列表，切分点，数量为输出张量数-1。

prototxt 文件中的层定义示例：

```
layer {
  name: "slice"
  type: "Slice"
  bottom: "tensor"
  ## Example of tensor with a shape N x 3 x 1 x 1
  top: "tensor_out1"
  top: "tensor_out2"
  top: "tensor_out3"
```

```
  slice_param {
    axis: 1
    slice_point: 1
    slice_point: 2
  }
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，包含输出张量名为“tensor_out1”的 Slice 层经过定点化后，参数字典中应有对应的 tensor_out1_frac 字段。

由于硬件限制，Slice 层有以下限制：

- Slice 层不允许作为神经网络的第一层；
- 要求输出张量与输入张量的位宽、小数点位置完全相同；
- 若输入为无符号数，输出张量也是无符号数。

6.13 Concat

Concat 层将多个输入 tensor 合并成一个输出 tensor。其对应的 Layer 对象中，bottom 的值为输入张量名的字符串列表，top 的值为字符串类型的输出张量名。

Concat 层有一个特有属性：

- axis - 整数类型对象，合并维度（目前只支持沿着通道维度切分，即 axis=1）；

prototxt 文件中的层定义示例：

```
layer {
  name: "concat"
  type: "Concat"
  bottom: "tensor_in1"
  bottom: "tensor_in2"
  top: "tensor_out"
  concat_param {
    axis: 1
  }
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为“tensor_out”的 Concat 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，Concat 层有以下限制：

- 输入张量名列表 bottom 中不允许有重复的张量名；
- Concat 层不允许作为神经网络的第一层。如需将其用于红绿蓝三通道图像转灰度图，请将此运算置于图像预处理过程，或者将该运算和首层卷积运算合并；

- 出现 bottom 中的张量，不能够出现在其它多输入层的 bottom 中，即一个张量最多只能作为一个多输入操作的输入，其中，多输入层包括 Concat、Eltwise。
- 要求输出张量与输入张量的位宽、小数点位置完全相同；
- 若输入为无符号数，输出张量也是无符号数。

6.14 Eltwise

将多个输入张量的对应元素做指定操作，指定操作包括：相加、相乘、取最大。

Eltwise 层对应的 Layer 对象中，bottom 的值为输入张量名的字符串列表，top 的值为字符串类型的输出张量名。

Eltwise 层有一个特有属性:

- operation – 字符串类型，用于指定逐元素操作的种类，‘sum’ 、‘max’ 、‘prod’（目前只支持‘sum’）。

prototxt 文件中的层定义示例:

```
layer {
    name: "eltwise"
    type: "Eltwise"
    bottom: "tensor_in1"
    bottom: "tensor_in2"
    top: "tensor_out"
    eltwise_param {
        operation: SUM
    }
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为”tensor_out”的 Eltwise 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，Eltwise 层有以下限制:

- 输入张量名列表 bottom 中不允许有重复的张量名。
- Eltwise 层不允许作为神经网络的第一层。
- 出现 bottom 中的张量，不能够出现在其它多输入层的 bottom 中，即一个张量最多只能作为一个多输入操作的输入，其中，多输入层包括 Concat、Eltwise。
- 配置为逐点相加（sum）时，输入张量的小数点位置需相同，输出张量的小数点位置值不能大于输入张量的小数点位置值。
- 配置为逐点取最大（max）时，输出张量和所有输入张量的小数点位置需相同。

6.15 Softmax

定点化不支持 Softmax，只支持 LogSoftmax。

若在解析时遇到网络最后一层是 Softmax，则打印一条警告，在模型打包时会自动把 Softmax 转成 LogSoftmax。注意，仅限“最后一层”，若出现在其它层，直接警告并退出。

输入一个向量$(x_0, x_1, \cdots, x_{n-1})$，Softmax 输出为$(p_0, p_1, \cdots, p_{n-1})$，其中：

$$p_i = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}}$$

Softmax 层无特有属性。

prototxt 文件中的层定义示例：

```
layers {
  name: "softmax"
  type: "Softmax"
  bottom: "tensor_in"
  top: "tensor_out"
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为”tensor_out”的 Softmax 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，Softmax 层有以下限制：

- Softmax 层只能置于不分组的 Convolution 层或者 InnerProduct 层之后；
- 暂时不支持 Softmax，若是网络最后的输出层，则会被转成 LogSoftmax。

6.16 LogSoftmax

只能在 Innerproduct、或者不分组的 Convolution 层（group=1）。

对 Softmax 的概率输出再做一个 log 操作。LogSoftmax 操作按通道维度进行，即对每一个 HW 维度坐标，取 C 个通道中对应元素构成向量，在该向量上进行 LogSoftmax 操作。

给定输入向量$(x_0, x_1, \cdots, x_{C-1})$，LogSoftmax 输出为$(q_0, q_1, \cdots, q_{C-1})$，其中：

$$q_i = \sigma(x_i) = \log\frac{e^{x_i}}{\sum_{j=0}^{C-1} e^{x_j}} = x_i - \log\sum_{j=0}^{C-1} e^{x_j}$$

在硬件实现时，第二项 log-sum-exp 项利用公式进行分解，

$$\log\left(e^a + e^b\right) = \max(a, b) + \log\left(1 + e^{-|a-b|}\right)$$

其中，$\log\left(1 + e^{-|a-b|}\right)$通过硬件查表进行快速计算。

LogSoftmax 层无特有属性。

prototxt 文件中的层定义示例：

```
layers {
  name: "logsoftmax"
  type: "LogSoftmax"
  bottom: "tensor_in"
  top: "tensor_out"
}
```

浮点模型在对应的参数字典中无对应字段。

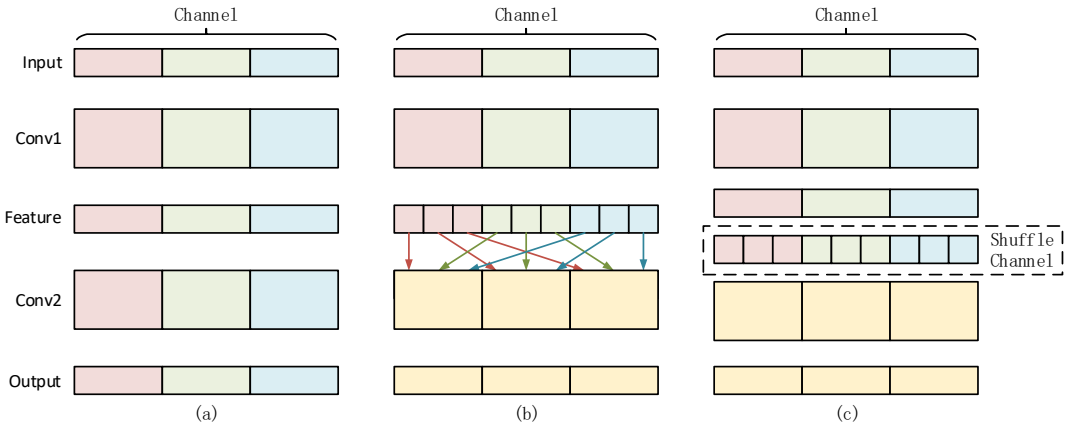
定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 LogSoftmax 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

对于 NCHW 格式的输入张量，记输入张量大小为(:, C, H, W)，输出张量大小也为(:, C, H, W)。由于硬件限制，定点 LogSoftmax 层的参数有以下限制：

- H 的值必须为 1；
- W 的值在 8 比特定点化时不能大于 32，16 比特定点化时不能大于 16；
- 在 8 比特位宽定点化时，LogSoftmax 层的输入和输出张量限定为 1 比特符号位、3 比特整数位、4 比特小数位，即小数点位置 tensor_in_frac=tensor_out_frac=4；
- 在 16 比特位宽定点化时，LogSoftmax 层的输入和输出张量限定为 1 比特符号位、3 比特整数位、12 比特小数位，即小数点位置 tensor_in_frac=tensor_out_frac=12。
- LogSoftmax 层只能置于不分组的 Convolution 层或者 InnerProduct 层之后。

6.17 ShuffleChannel

通道混洗操作如下图示例所示：图 a）两个连续的卷积层有相同数量的分组，这样就会导致每个输出通道仅仅来自于输入通道的对应分组的一小部分，这样导致能够表述特性有局限性。图 b）一个简单的改进就是第二层卷积 Conv2 之前，对其输入特征图做一次交织，每个分组分成几个子分组，然后将来自不同分组的子分组组合起来作为 Conv2 的一个分组的输入，如此 Conv2 的每一个分组都有来自输入张量的所有通道分组的特征。图 c）以上操作等效于在 Conv2 之前插入一个 ShuffleChannel 层。



ShuffleChannel 层有一个特有属性：

- group – 整数类型，卷积分组数，请注意分组数需要整除通道数目。

prototxt 文件中的层定义示例：

```
layers {
  name: "shuffle"
  type: "ShuffleChannel"
  bottom: "tensor_in"
  top: "tensor_out"
  shuffle_channel_param {
   group: 4
  }
}
```

浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为"tensor_out"的 ShuffleChannel 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，ShuffleChannel 层有以下限制：

- ShuffleChannel 层只能置于不分组的 Convolution 层或者 InnerProduct 层之后；
- 要求输出张量与输入张量的位宽、小数点位置完全相同；
- 若输入为无符号数，输出张量也是无符号数。

6.18 ScaleByTensor

该层是 SENet 中的重要组成部分，运算规则和 Scale 层类似，区别在于该层使用网络中某层的输出作为 Scale 的权重且该层无偏置。

ScaleByTensor 层无特有属性。

prototxt 文件中的层定义示例：

```
layers {
  name: "logsoftmax"
  type: "LogSoftmax"
  bottom: "tensor_in"
  bottom: "tensor_weight"
  top: "tensor_out"
}
```

其中 tensor_weight 表示 scale 运算的权重且 tensor_in 和 tensor_weight 的顺序不得颠倒。

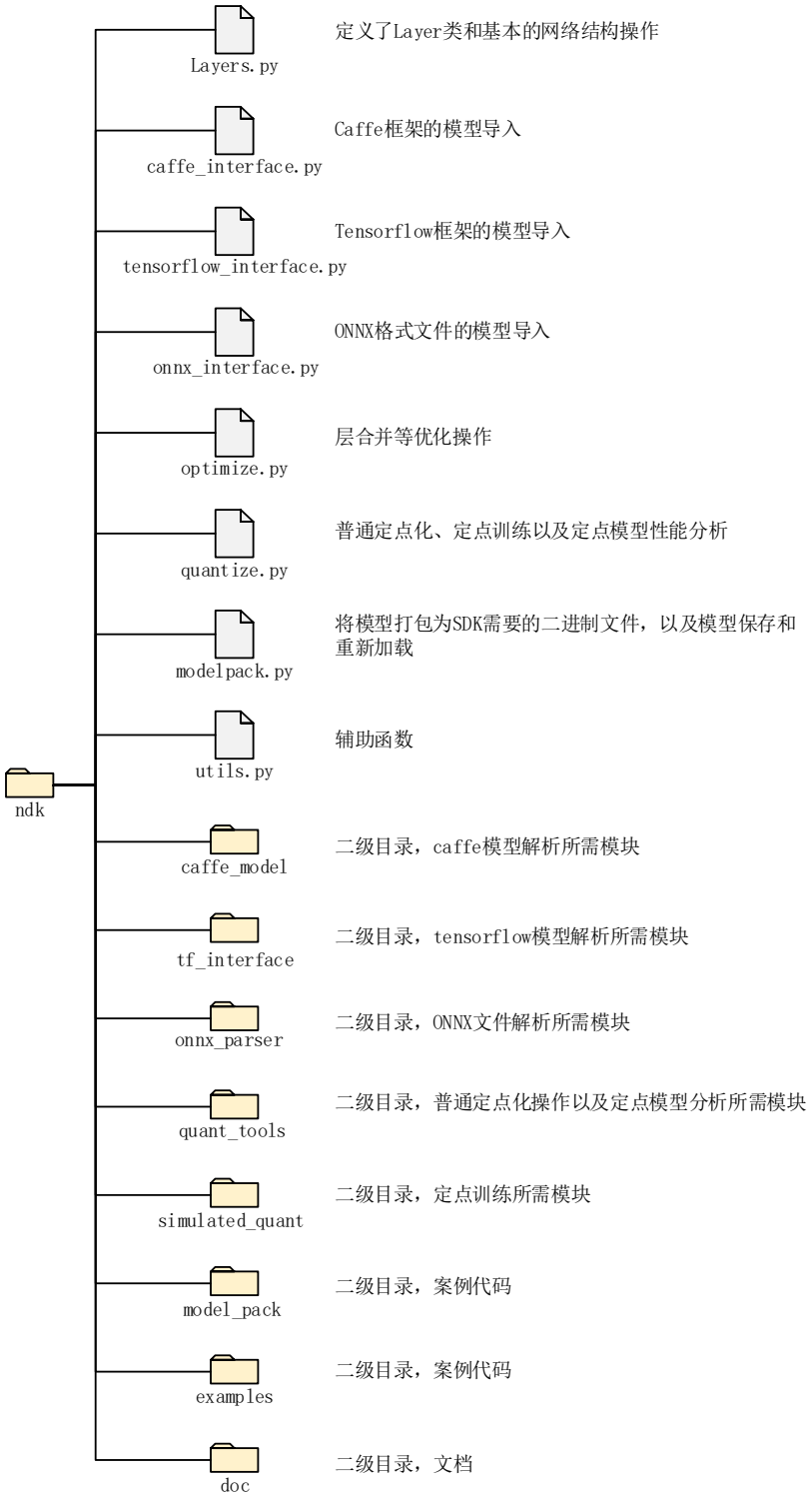
浮点模型在对应的参数字典中无对应字段。

定点化模型对应的参数字典中存有输出张量的量化参数。例如，输出张量名为”tensor_out”的 ScaleByTensor 层经过定点化后，参数字典中应有对应的 tensor_out_frac 字段。

由于硬件限制，ShuffleChannel 层有以下限制：

- 输出只能是有符号数。
- 定点化参数中：
  - 8 位定点化，0 ≤ tensor_in_frac + tensor_weight_frac - tensor_out_frac ≤ 15；
  - 16 位定点化，0 ≤ tensor_in_frac + tensor_weight_frac - tensor_out_frac ≤ 31；

七、 代码存放路径



NDK 各个模块的代码存放路径如上图所示。

除去功能模块，在二级目录 example 提供了 NDK 的使用样例代码；在 doc 目录下存有 NDK 的技术文档。

八、 Python 生成器用法简介

在 Python 中，由于受到内存的现在，列表容量是受限的。如果创建一个包含一个亿个元素列表，会占用很大的存储空间，若用户只想访问列表前面几个元素，则列表后面绝大多数元素占用的空间就都白白浪费了。

为了解决以上问题，Python 引入了一中“一边循环，一边计算”的机制，即当用户需要使用某个对象时，Python 才根据事先设计好的规则开辟内存空间创建这个对象供用户使用，而不是像列表一样事先将所有的对象都创建完毕之后再提供给用户使用。这种机制在 Python 中称为生成器（generator）。

生成器是一个特殊的程序，可以被用作控制循环的迭代行为，Python 中生成器是迭代器的一种，使用 yield 返回值函数，每次调用 yield 会暂停，可以使用 next()函数和 send()函数恢复生成器。

生成器类似于返回值为数组的一个函数，这个函数可以接受参数，可以被调用，但是，不同于一般的函数会一次性返回包括了所有数值的数组，生成器一次只能产生一个值，这样消耗的内存数量将大大减小，而且允许调用函数可以很快的处理前几个返回值，因此生成器看起来像是一个函数，但是表现得却像是迭代器。

8.1 创建生成器（generator）

创建一个 generator 的方法很多，下面介绍一种简单的方法，例如：把一个列表生成式的[ ]中括号改为（）小括号即可创建一个 generator。

```
#列表生成式
lis = [x*x for x in range(10)]
print(lis)
#生成器
generator_ex = (x*x for x in range(10))
print(generator_ex)
```

运行结果:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
<generator object <genexpr> at 0x000002A4CBF9EBA0>
```

创建 lis 和 generator_ex 的区别从表面看就是[ ]和（ ），但结果却不一样，前者打印出来是列表（因为是列表生成式），后打印出来却是**<generator object <genexpr> at 0x000002A4CBF9EBA0>**。可以通过 **next（）**方法获得 generator 的下一个返回值，打印出 generator_ex 的每一个元素:

```
#生成器
generator_ex = (x*x for x in range (10))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
```

```
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
```

运行结果:

```
0
1
4
9
16
25
36
49
64
81
Traceback (most recent call last):
File "列表生成式.py", line 42, in <module>
print(next(generator_ex))
StopIteration
```

从上面的结果可以得出 generator 保存的是算法，每次调用 **next(generaotr_ex)**就计算出列表的下一个元素的值，直到计算出最后一个元素时，抛出 StopIteration 的错误。也可以使用 for 循环进行遍历，因为 generator 也是可迭代对象:

```
#生成器
generator_ex = (x*x for x in range(10))
for i in generator_ex:
print(i)
```

运行结果:

```
0
1
4
9
16
25
36
49
64
81
```

若有复杂的推算算法，用类似列表生成式的 for 循环无法实现，则可以用函数来实现。比如著名的斐波那契数列，除第一个和第二个数外，任何一个数都可以由前两个相加得到：1，1，2，3，5，8，12，21，34.....

斐波那契数列用函数很容易打印出来:

```
#fibonacci 数列
def fib(max):
   n,a,b =0,0,1
      while n < max:
         a,b =b,a+b
         n = n+1
         print(a)
   return 'done'
print(fib(10))
```

fib 函数是定义了斐波拉契数列的推算规则，可以从第一个元素开始，逐个推算出后续任意的元素，这种逻辑与 generator 非常类似。在需要打印的时候，通过关键字 **yield** 暂停推算过程，同时将当前数值返回。

```
def fib(max):
   n,a,b =0,0,1
   while n < max:
         yield b
         a,b =b,a+b
         n = n+1
   return 'done'
print(fib(10))
```

上面的返回值 **fib(10)**不再是一个值，而是一个生成器，上面的运行结果如下：

```
<generator object fib at 0x000001C03AC34FC0>
```

generator 和函数的执行流程：

- 函数是顺序执行的，遇到 **return** 语句或者最后一行函数语句就返回。
- 变成 generator 的函数，在每次调用 **next()**的时候执行，遇到 **yield** 语句返回，再次被 **next**（）调用时候从上次的返回 **yield** 语句处急需执行，也就是用多少，取多少。

Python 提供了两种基本的方式：生成器表达式和生成器函数。

8.2 生成器表达式

生成器表达式来源于迭代和列表解析的组合，生成器和列表解析类似，是使用圆括号。返回一个对象，这个对象只有在需要的时候才产生结果。

```
# 列表解析生成列表
[ x ** 3 for x in range(5)]
[0, 1, 8, 27, 64]

# 生成器表达式
```

```
(x ** 3 for x in range(5))
<generator object <genexpr> at 0x000000000315F678>
# 两者之间转换
list(x ** 3 for x in range(5))
[0, 1, 8, 27, 64]
```

8.3 生成器函数

和普通函数一样用关键字 **def** 定义，利用关键字 **yield** 一次性返回一个结果，同时阻塞推算过程，再被 **next()**调用时重新开始。

一般的函数在执行完毕之后会返回一个值并退出，但是生成器函数会自动挂起，然后重新拾起急需执行，它利用 **yield** 关键字关起函数，给调用者返回一个值，同时保留了当前足够多的状态，可以使函数继续执行，生成器和迭代协议是密切相关的，迭代器都有一个**__next__()__**成员方法，这个方法要么返回迭代的下一项，要么引起异常结束迭代。

```
# 函数使用了关键字 yield 之后，函数名+()返回的就是一个生成器对象
# return 在生成器中代表生成器的中止，直接报错
# next 的作用是唤醒并继续执行
# send 的作用是唤醒并继续执行，发送一个信息到生成器内部
'''生成器'''
def create_counter(n):
   print("create_counter")
   while True:
      yield n
      print("increment n")
      n +=1

gen = create_counter(2)
print(gen)
print(next(gen))
print(next(gen))
```

运行结果:

```
<generator object create_counter at 0x0000023A1694A938>
create_counter
2
increment n
3
Process finished with exit code 0
```

九、 实例：基于 LeNet-5 实现 MNIST 手写数字识别

本节内容通过一个具体的例子——基于 LeNet5 的 MNIST 手写数字识别，力图对 NDK 使用过程中的各个环节作出一个完整呈现。

本例完整源码见 ndk/example/example_lenet5.py。

9.1 编写 MNIST 生成器函数

MNIST 是一个入门级的计算机视觉数据集，官网：http://yann.lecun.com/exdb/mnist/。

MNIST 数据集包含了 60000 张的训练图片和 10000 张的测试图片，每张图片是 28 x 28 像素，即 784 个像素点。下载后的 MNIST 数据集主要是下面的四个文件：

- train-images-idx3-ubyte.gz：包含 60,000 个训练样本图像；
- train-labels-idx1-ubyte.gz：包含 60,000 个训练样本标签；
- t10k-images-idx3-ubyte.gz：包含 10,000 个测试样本图像；
- t10k-labels-idx1-ubyte.gz：包含 10,000 个测试样本标签。

为了方便使用，同时也为了在定点化的时候提供统计数据分布用的数据，需要将产生样本 batch 的过程封装成一个 Python 生成器函数。每一次对这个生成器调用 `next()` 方法，都会返回一个字典类型的结果，包含了一个 batch 的神经网络输入的图像张量和对应的希望神经网络输出的张量。

NDK 关于数据样本生成器的要求请参见《数据样本生成器》；Python 生成器语法介绍请参考《Python 生成器用法简介》。

由于 MNIST 数据集比较小，可以先一次性把整个二进制文件读进内存。

```
data = []
if use_test_set:
    with open(os.path.join(mnist_dirname,
                           't10k-images.idx3-ubyte'),'rb') as f:
        buf_image = f.read()
    with open(os.path.join(mnist_dirname,
                           't10k-labels.idx1-ubyte'),'rb') as f:
        buf_label = f.read()
else:
    with open(os.path.join(mnist_dirname,
                           'train-images.idx3-ubyte'),'rb') as f:
        buf_image = f.read()
    with open(os.path.join(mnist_dirname,
                           'train-labels.idx1-ubyte'),'rb') as f:
        buf_label = f.read()
```

对于图像文件，需要跳过最前面 16 字节（magic number、图像数、图像高度、图像宽度各 32 比特），每张图像为连续的 28*28=784 个字节；对于标签文件，需要跳过最前面 8 字节（magic number、标签数各 32 比特），每个标签数据为 1 字节。

```
image_index_offset = struct.calcsize('>IIII')
label_index_offset = struct.calcsize('>II')
image_sample_size = struct.calcsize('>784B')
label_sample_size = struct.calcsize('>B')

num_samples = int( (len(buf_image)-image_index_offset)
                                        /image_sample_size )
```

根据需要，顺序或者随机抽取当前 batch 的样本。同时为了让生成器函数能够产生无限多的样本（训练时可能需要非常多的样本），因此样本提取的代码放在无限循环 `while True` 里面：

```
sample_idx = -1
while True:
   input_list = []
   output_list = []
   for _ in range(batch_size):
      if random_order:
         sample_idx = np.random.randint(num_samples)
      else:
         sample_idx += 1
         if sample_idx >= num_samples:
            sample_idx = 0

      # draw the selected sample from the data set
      sample_data = np.array(struct.unpack_from('>784B',
                             buf_image,
                             image_index_offset
                              + sample_idx*image_sample_size))
      sample_label = np.array(struct.unpack_from('>B',
                              buf_label,
                              label_index_offset
                              + sample_idx*label_sample_size))

      # do any data pre-processing here!
      …
```

将这一个 batch 的数据和标签处理成神经网络的输入和输出张量的尺寸，格式为 NCHW，并用 `yield` 关键字返回。

如果需要对输入图像进行任何预处理操作，比如将 0-255 的像素值归一化到 0-1 的浮点数，再比如输入图像大小的调整，这些预处理操作都需要在数据生产器函数中完成。

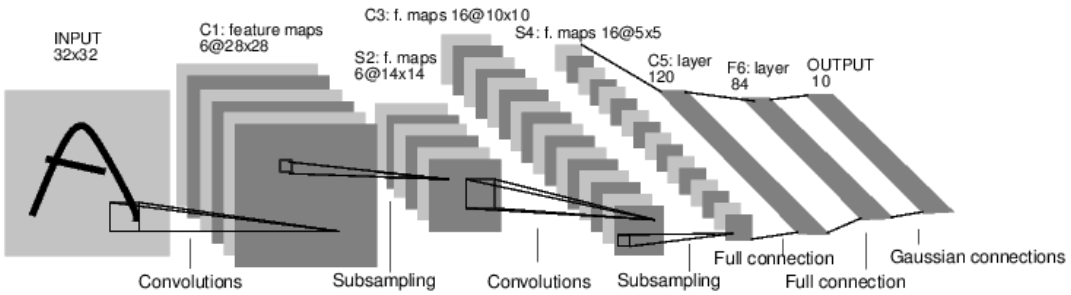
```
# do any data pre-processing here!
# e.g. padding the image to 32x32 and normalizing to 0-1
net_input = np.zeros((1,32,32)) # CHW format
net_input[:, 2:30, 2:30] = 1./256 * np.array(
                                sample_data.reshape((1,28,28)),
                                dtype=np.float64)
# e.g. translate label to one-hot vector as output
net_output = np.zeros((10,1,1))
net_output[sample_label, 0, 0] = 1.0
input_list.append(net_input)
output_list.append(net_output)

yield {'input': np.array(input_list),
       'output': np.array(output_list)}
```

完整代码见 ndk/examples/data_generator_mnist.py。在 ndk/example/mnist 路径下，也保存了一份从 MNIST 官网下载的二进制文件。

9.2 训练浮点模型

LeNet-5 是一种用于手写体字符识别的非常高效的卷积神经网络，提出自论文 Gradient-Based Learning Applied to Document Recognition [Yann, 98']。



LeNet-5 结构比较简单，由 2 个卷积层、2 个池化层和 3 个全连接层构成，激活函数都采用 ReLU。

在 TensorFlow 框架下实现如下：

```
def LeNet5(input):
    with tf.variable_scope('conv1'):
        conv1_weights = tf.get_variable(
            'weight', [5, 5, 1, 6],
            initializer=tf.truncated_normal_initializer(stddev=0.1)
```

```
        )
    conv1_biases = tf.get_variable('bias', [6],
        initializer=tf.constant_initializer(0.1)
        )
    conv1 = tf.nn.conv2d(input, conv1_weights,
        strides=[1, 1, 1, 1], padding='VALID', name='conv'
        )
    conv1_out= tf.nn.bias_add(conv1, conv1_biases,
        name='conv_out'
        )
    relu1 = tf.nn.relu(conv1_out, name='relu')
    pool1 = tf.nn.max_pool(relu1, ksize=[1, 2, 2, 1],
        strides=[1, 2, 2, 1], padding='VALID', name='max_pool'
        )

with tf.variable_scope('conv2'):
    conv2_weights = tf.get_variable(
        'weight', [5, 5, 6, 16],
        initializer=tf.truncated_normal_initializer(stddev=0.1)
        )
    conv2_biases = tf.get_variable('bias', [16],
        initializer=tf.constant_initializer(0.1)
        )
    conv2 = tf.nn.conv2d(pool1, conv2_weights,
        strides=[1, 1, 1, 1], padding='VALID', name='conv'
        )
    conv2_out = tf.nn.bias_add(conv2, conv2_biases,
        name='conv_out'
        )
    relu2 = tf.nn.relu(conv2_out, name='relu')
    pool2 = tf.nn.max_pool(relu2, ksize=[1, 2, 2, 1],
        strides=[1, 2, 2, 1], padding='VALID', name='max_pool'
        )

    pool_shape = pool2.get_shape().as_list()
    nodes = pool_shape[1] * pool_shape[2] * pool_shape[3]
    reshaped = tf.reshape(pool2, [-1, nodes])

with tf.variable_scope('fc1'):
    fc1_weights = tf.get_variable(
        'weight', [nodes, 120],
        initializer=tf.truncated_normal_initializer(stddev=0.1)
        )
    fc1_biases = tf.get_variable('bias', [120],
```

```
            initializer=tf.constant_initializer(0.1)
            )
        fc1 = tf.matmul(reshaped, fc1_weights) + fc1_biases
        fc1_out = tf.identity(fc1, name='fc_out')
        fc1_relu = tf.nn.relu(fc1_out, name='relu')

    with tf.variable_scope('fc2'):
        fc2_weights = tf.get_variable(
            'weight', [120, 84],
            initializer=tf.truncated_normal_initializer(stddev=0.1)
            )
        fc2_biases = tf.get_variable('bias', [84],
            initializer=tf.constant_initializer(0.1)
            )
        fc2 = tf.matmul(fc1_relu, fc2_weights) + fc2_biases
        fc2_out = tf.identity(fc2, name='fc_out')
        fc2_relu = tf.nn.relu(fc2_out, name='relu')

    with tf.variable_scope('fc3'):
        fc3_weights = tf.get_variable(
            'weight', [84, 10],
            initializer=tf.truncated_normal_initializer(stddev=0.1))
        fc3_biases = tf.get_variable('bias', [10],
            initializer=tf.constant_initializer(0.1)
            )
        fc3 = tf.matmul(fc2_relu, fc3_weights) + fc3_biases
        fc3_out = tf.identity(fc3, name='fc3_out')

    return fc3_out
```

定义完网络之后，使用常用的方法训练。本例中，我们使用了 Adam（学习率 1e-4）以交叉熵为代价函数，每一个 batch 包含 MNIST 训练集里随机的 32 张图，训练 200000 步。训练得到的模型在 10000 张图的 MNIST 测试集的识别正确率为 99.04%。

注意，TensorFlow 的默认数据格式为 NHWC 格式，如果由前一小节定义的数据生成器函数获取训练样本，则要注意从 NCHW 格式转为 NHWC 格式。

```
net_input = tf.placeholder(tf.float32,
                shape=[None, 32, 32, 1],
                name='net_input'
                ) # NHWC format
net_output = tf.identity(LeNet5(net_input), name="net_output")
true_output = tf.placeholder(tf.float32, shape=[None, 10])

# codes for training
```

```
    …
    batch = next(g)
    batch_in = batch['input']
    batch_out = batch['output']
    batch_in = batch_in.transpose([0,2,3,1]) # NCHW to NHWC
    batch_out = batch_out.transpose([0,2,3,1]).reshape((-1, 10))
    …
```

最后，将训练好的模型导出为 pb 文件。

保存 pb 文件时需要将计算图中的变量转化为常量，以便用于推理阶段。这里我们使用函数 `tf.graph_uitl.convert_variables_to_constants`。该函数需要指定欲保存网络的输出节点名称（区别于张量名）。注意推理阶段不需要损失函数计算、更新梯度等节点，因此这里我们指定最后一层全连接的结果为输出节点。

```
# freeze the graph and save to pb file
constant_graph = tf.graph_util.convert_variables_to_constants(
                     sess=sess,
                     input_graph_def=sess.graph.as_graph_def(),
                     output_node_names=['net_output']
                     )
with tf.gfile.GFile(fname_pb, "wb") as f:
    f.write(constant_graph.SerializeToString())
```

至此，我们已经得到了一个性能足够好用于手写体识别的神经网络。下面，我们将演示如何通过 NDK 将这个神经网络移植到物奇芯片平台。

9.3 导入模型并查看参数

使用 NDK 函数的第一步是导入将训练好的模型。读取的模型会保存为一个 Layer 对象列表和参数字典。

```
# load the model from pb file
layer_list, param_dict = ndk.tensorflow_interface.load_from_pb(
                                                  r'lenet5_model.pb'
                                                  )
```

然后检查一下网络结构以及参数大小等是否超出了 NPU 的限制。

建议尽可能地多调用 `check_layers` 函数进行参数检查，以便在定点化过程中及时发现不被硬件支持的模型结构和参数。

```
# check the validity of the parameters
ndk.layers.check_layers(layer_list, param_dict)
```

在导入模型之后，我们尝试调用层合并函数将网络中包含的 Scale/BatchNorm/Bias 层合并到前后的卷积或者全连接层里去，以此来减少网络的层数，从而达到提高硬件执行效率的目的。

对于 LeNet-5 来说，因为不含有 Scale/BatchNorm/Bias 层，因此层合并操作返回的层列表和参数字典相对于输入的不会有变化。

```
# merge layers if possible
layer_list, param_dict = ndk.optimize.merge_layers(layer_list,
                                                   param_dict)
```

我们可以获取并打印查看模型中所有层的名字：

```
# show all the layer names
print('\nLayer name list:')
print(ndk.layers.get_layer_name_list(layer_list))
```

执行结果为：

```
Layer name list:
['net_input', 'conv1/conv', 'conv1/relu', 'conv1/max_pool',
'conv2/conv', 'conv2/relu', 'conv2/max_pool', 'fc1/dense',
'fc1/relu', 'fc2/dense', 'fc2/relu', 'fc3/dense']
```

也可以获取并打印模型中所有张量的名字：

```
# show all the tensor names
print('\nTensor name list:')
print(ndk.layers.get_tensor_name_list(layer_list,
                                      feature_only=False))
```

执行结果为：

```
Tensor name list:
['net_input:0', 'conv1/conv_out:0', 'conv1/relu:0',
'conv1/max_pool:0', 'conv2/conv_out:0', 'conv2/relu:0',
'conv2/max_pool:0', 'fc1/fc_out:0', 'fc1/relu:0', 'fc2/fc_out:0',
'fc2/relu:0', 'net_output:0', 'conv1/conv_weight',
'conv1/conv_bias', 'conv2/conv_weight', 'conv2/conv_bias',
'fc1/dense_weight', 'fc1/dense_bias', 'fc2/dense_weight',
'fc2/dense_bias', 'fc3/dense_weight', 'fc3/dense_bias']
```

其中，卷积、全连接以及 Scale 层的权重和偏置以“层名称_weight”和“层名称_bias”的格式给出。

或者单独获取网络最后输出张量的名字：

```
# network output tensor name
output_tensor = ndk.layers.get_network_output(layer_list)
```

我们可以在命令行窗口中打印出所有层的属性以及张量名：

```
# show the layer properties
print('\nlayers:')
for layer in layer_list:
   print(layer)
```

如果网络规模比较大，建议通过 `ndk.modelpack.save_to_file` 函数导出为 NDK 格式的 prototxt 文件（`param_dict` 参数可以不传），然后用文本编辑器查看。

```
# export layer_list to prototxt file
ndk.modelpack.save_to_file(
                           layer_list=layer_list,
                           fname_prototxt='loaded_model.prototxt'
                           )
```

9.4 模型定点化

NPU 计算要求模型量化为 8 比特或者 16 比特。NDK 提供了两种不同的模型定点化方法。多数情况下，通过调整普通定点化工具的参数就可以得到不错的结果，但若遇到模型在经过普通定点化后性能损失较大，NDK 还提供了根据定点化参数重新进行网络权重/偏置训练。

考虑到模型定点化过程中，会需要反复调整参数。建议：

- 在每次调用定点化后将得到的定点模型通过 `ndk.modelpack.save_to_file` 函数保存为 NDK 格式的 prototxt 和 npz 文件；
- 在需要分析模型性能、调整参数、或者最终导出为二进制文件时，再通过 `ndk.modelpack.load_from_file` 函数从 prototxt 和 npz 文件中读取。

9.4.1 普通定点化

普通定点化工具通过一小部分输入数据样本，统计各个特征张量的分布，据此进行定点参数的确定。

定点化工具需要一部分的数据样本用于统计特征张量分布，因此首先需要产生一个提供数据的生成器函数。如下所示，每个 batch 包含 32 张从训练集合里随机抽取的图。

```
g_train = mnist.data_generator_mnist(mnist_dirname='mnist',
                                     batch_size=32,
                                     random_order=True,
                                     use_test_set=False
                                     )
```

将导入的层列表和参数字典送给定点化工具函数 `ndk.quantize.quantize_model`，设定量化位宽为 8 比特，采用快速定点方案（`aggressive=True`）。数据预处理阶段采用默认的 20 个 batch，定点参数确定阶段使用默认的 1000 个 batch 的数据。

```
quant_layer_list, quant_param_dict = ndk.quantize.quantize_model(
                                     layer_list=layer_list,
                                     param_dict=param_dict,
                                     bitwidth=8,
                                     data_generator=g_train,
                                     aggressive=True,
                                     num_step_pre=20,
                                     num_step=1000
                                     )
```

定点化之后，在参数字典中会增加定点参数的字段。

将定点模型保存为 prototxt 和 npz 文件，以备进一步参数调整或者性能分析。

```
ndk.modelpack.save_to_file(
                 layer_list=quant_layer_list,
                 fname_prototxt='example_lenet5_quant.prototxt',
                 param_dict=quant_param_dict,
                 fname_npz='example_lenet5_quant.npz'
                 )
```

9.4.2 定点模型训练

除了使用普通定点化工具对浮点模型进行定点化，也可以使用定点模型训练工具。区别在于定点模型训练会根据网络的输入和目标输出，在定点约束下，对模型参数（权重、偏置等）进行调整。

```
quant_layer_list, quant_param_dict = \
             ndk.quantize.quantize_model_with_training(
                       layer_list=layer_list,
                       bitwidth=8,
                       data_generator=g_train,
                       param_dict=param_dict,
                       usr_param_dict=None,
                       log_dir='log',
                       loss_fn=tf.losses.softmax_cross_entropy,
                       optimizer=tf.train.AdamOptimizer(0.00001),
                       num_step_stats=200,
                       num_step_train=1000
                       )
```

9.5 模型性能分析

NDK 提供了分析各个张量分布以及对比定点化前后性能的工具。用户也可以通过运行模型，对模型的输出（也可以是任何中间层的输出张量）进行自定义的分析，比如识别正确率等。

在进行分析之前，用 `load_from_file` 函数读取之前保存的定点模型参数。

```
quant_layer_list, quant_param_dict = \
        ndk.modelpack.load_from_file(
                  fname_prototxt='example_lenet5_quant.prototxt',
                  fname_npz='example_lenet5_quant.npz
                  )
```

生成一个 MNIST 测试集生成器函数，每个 batch 包含 100 张图，顺序遍历整个测试集。

```
g_test = mnist.data_generator_mnist( mnist_dirname='mnist',
                                     batch_size=100,
                                     random_order=False,
                                     use_test_set=True)
```

9.5.1 分析张量分布

首先，我们来分析一下原先浮点模型。除了网络的输入输出，例程中还要求关注第一个卷积层的权重、偏置，以及第一个卷积层和第一个池化层的输出张量。

用 100 个 batch 的数据进行统计（由于 MNIST 测试集合总共只有 10000 张图，100 个 batch 就是整个测试集），除去一般的统计信息之外，还要求程序统计这些张量的概率分布：概率分布统计用的各个 bin 中心值根据各自动态范围决定（不设定最大张量值 `max_abs_val=None`），bin 的总数为 51 个。

指定网络按浮点参数执行，配置 `quant=False`。

```
float_stats =  ndk.quantize.analyze_tensor_distribution(
                    layer_list=quant_layer_list,
                    param_dict=quant_param_dict,
                    target_tensor_list = ['net_input:0',
                                          'conv1/conv_weight',
                                          'conv1/conv_bias',
                                          'conv1/conv_out:0',
                                          'conv1/max_pool:0',
                                          'net_output:0'],
                    output_dirname=os.path.join('res', 'float'),
                    data_generator=g_test,
                    quant=False,
                    num_batch=100,
```

```
                    num_bin=51,
                    max_abs_val=None
                    )
```

执行以上代码之后，能在参数 `output_dirname` 指定的 res/float 路径下找到一个 stats.csv 和一个 prob.csv。

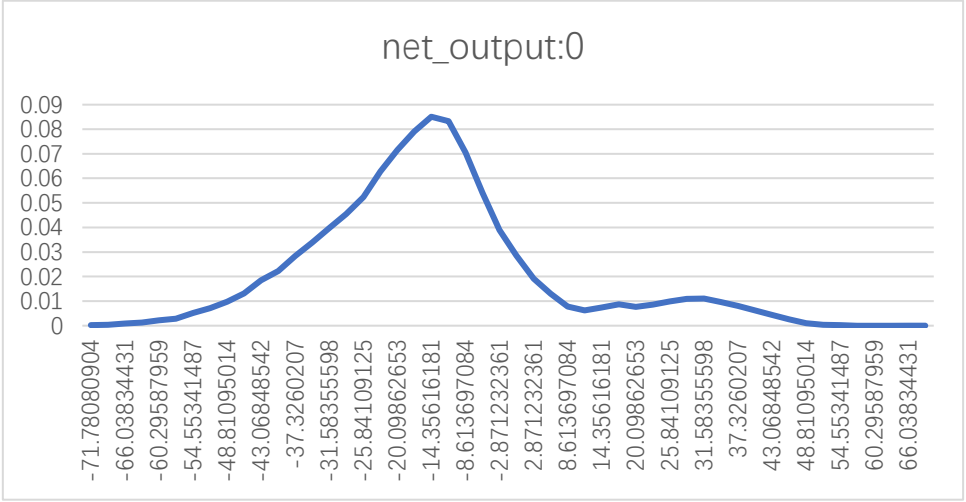
统计信息文件 stats.csv 给出了各个目标张量在浮点模型下的均值、方差和最大最小值，文件内容格式如下所示：

	A	B	C	D	E	F	G
1		net_input:0	conv1/conv_weight	conv1/conv_bias	conv1/conv_out:0	conv1/max_pool:0	net_output:0
2	avg	0.101060181	0.049684629	0.064799212	0.228473049	0.430579097	-11.54590259
3	max	0.99609375	0.513549149	0.150768593	3.729869354	3.729869354	66.49388064
4	min	0	-0.798605561	-0.015115714	-4.20629682	0	-76.72393047
5	var	0.096782031	0.061040483	0.003242022	0.576022528	0.867772381	646.5432522

数据分布文件 prob.csv 给出了各个目标张量在浮点模型下的概率分布。每相邻的两列对应一个目标张量，其中奇数列给出的是各个 bin 的中心值，偶数列给出的是对应的概率值。文件内容格式如下所示：

	A	B	C	D	E	
1	net_input:0_val	net_input:0_prob	conv1/conv_weight_val	conv1/conv_weight_prob	conv1/conv_bias_val	conv1/
2	0	0.853356348	-0.798605561	0.006666667	-0.150768593	
3	0.019921875	0.003118066	-0.766661339	0	-0.144737849	
4	0.03984375	0.00275459	-0.734717116	0	-0.138707106	
5	0.059765625	0.002462598	-0.702772894	0	-0.132676362	
6	0.0796875	0.00258584	-0.670828671	0.006666667	-0.126645618	
7	0.099609375	0.002436523	-0.638884449	0	-0.120614874	
8	0.11953125	0.002086523	-0.606940227	0	-0.114584131	
9	0.139453125	0.001845605	-0.574996004	0	-0.108553387	
10	0.159375	0.001755664	-0.543051782	0.006666667	-0.102522643	
11	0.179296875	0.0016375	-0.511107559	0.006666667	-0.096491899	
12	0.19921875	0.001616406	-0.479163337	0	-0.090461156	
13	0.219140625	0.001671484	-0.447219114	0.013333333	-0.084430412	
14	0.2390625	0.001542773	-0.415274892	0.006666667	-0.078399668	
15	0.258984375	0.001856934	-0.383330669	0.013333333	-0.072368925	
16	0.27890625	0.001483691	-0.351386447	0.006666667	-0.066338181	
17	0.298828125	0.001583203	-0.319442225	0.02	-0.060307437	
18	0.31875	0.001535156	-0.287498002	0.02	-0.054276693	

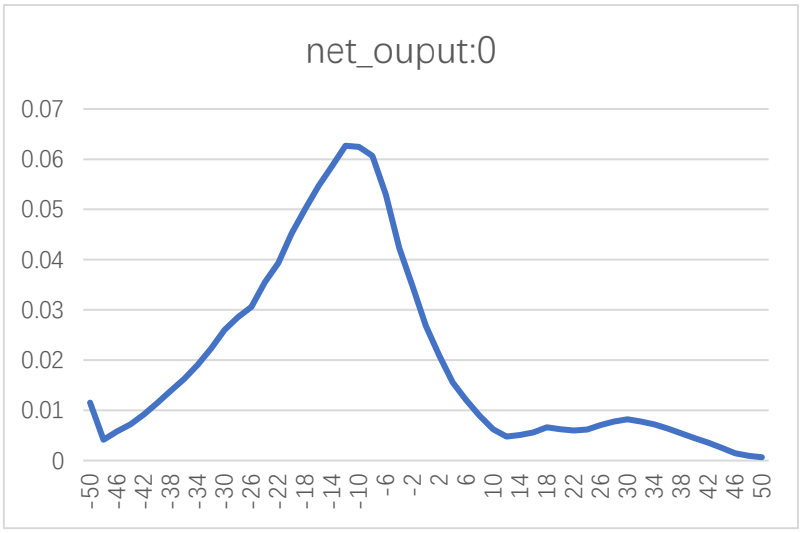
利用 excel 的绘图功能就可以绘制各个张量的概率分布曲线，或者也可以将 csv 数据导入到 matlab 或者利用 python 环境中利用相应的数据分析工具进行进一步分析。



通过更改 `quant=True` 并且配置 `bitwidth=8`，重新运行就可以得到这些指定张量在执行定点模型时的统计值。

```
quant_stats =  ndk.quantize.analyze_tensor_distribution(
                    layer_list=quant_layer_list,
                    param_dict=quant_param_dict,
                    target_tensor_list = ['net_input:0',
                                          'conv1/conv_weight',
                                          'conv1/conv_bias',
                                          'conv1/conv_out:0',
                                          'conv1/max_pool:0',
                                          'net_output:0'],
                    output_dirname=os.path.join('res', 'quant'),
                    data_generator=g_test,
                    quant=True,
                    bitwidth=8,
                    num_batch=100,
                    hw_aligned=True,
                    num_bin=51,
                    max_abs_val=50,
                    log_on=False)
```

运行以上代码之后，统计结果保存在 `output_dirname` 指定的 res/quant 路径下。由于配置了 `max_abs_val=50`，概率分布统计用的各个 bin 中心值的确定直接从-50 到 50 区间内均匀地取 `num_bin=51` 个值。



9.5.2 浮点和定点模型比较

通过调用 `compare_float_and_quant` 函数，可以方便地对浮点和定点模型性能的性能进行比较。注意，用于比较的参数字典必须已是经过定点化、包含有定点化参数的。

```
compare_dict = ndk.quantize.compare_float_and_quant(
                    layer_list=quant_layer_list,
                    param_dict=quant_param_dict,
                    target_tensor_list=['net_input:0',
                                        'conv1/conv_weight',
                                        'conv1/conv_bias',
                                        'conv1/conv_out:0',
                                        'conv1/max_pool:0',
                                        'net_output:0'],
                    bitwidth=8,
                    output_dirname='res',
                    data_generator=g_test,
                    num_batch=100,
                    hw_aligned=True,
                    num_bin=51,
                    log_on=False
                    )
```

运行以上代码之后，结果保存在`output_dirname`指定的res路径下的comp_results.csv，文件内容如下：

	A	B	C	D	E	F	G
1		net_input:0	conv1/conv_weight	conv1/conv_bias	conv1/conv_out:0	conv1/max_pool:0	net_output:0
2	Flt-Pnt Mean Sqr Val	0.086568871	0.06350904	0.00744096	0.523822594	0.682374022	513.2353855
3	Fix-Pnt Mean Sqr Val	0.086568871	0.063549805	0.007399919	0.426039294	0.522291335	336.4986125
4	Mean Abs Error	0	0.001590668	0.000193944	0.027904717	0.039509387	3.98402909
5	Max Abs Error	0	0.003878504	0.00044246	2.20629682	1.745494354	19.85520416
6	Mean Sqr Error	0	3.49E-06	6.60E-08	0.015254802	0.025625037	25.12098836
7	Quant SNR (dB)	89.37361756	42.59933297	50.51223197	15.35777639	14.25357986	13.10279869
8	Top1 Error Rate	0	0	0	0.038820281	0.073152041	0.0091
9	KL Divergence	0	0.308707392	8.235839583	1.756793504	2.291387791	0.20507793
10	JS Divergence	0	0.009129513	0.166666667	0.040703273	0.040891126	0.026615048
11	Sample Num	10240000	150	6	47040000	11760000	100000

分析结果给出了目标张量在浮点/定点模型里的均方值、两者的平均绝对值误差、最大绝对值误差、均方误差、量化信噪比、Top1 序号不一致比率、Kullback-Leibler 散度、Jensen-Shannon 散度、以及统计 KLD/JSD 时所用的样本数量。

值得注意的是，由于 KLD 和 JSD 在计算时候依赖统计得到的概率分布，因此在样本数量较小时，KLD 和 JSD 的参考价值就会受限。

例如，在上图结果中，我们发现 conv1/conv 卷积层偏置的量化信噪比达到 50dB，但在定点化前后 KL 散度和 JS 散度值却比较大。查看统计用的样本数（Sample Num）发现，由于该层偏置只有 6 个值，所以导致分布统计得不准确，此时 KLD 和 JSD 值意义不大。

9.5.3 用户自定义操作

除了使用以上两种分析工具，NDK 用户还可以直接执行浮点/定点模型计算过程，获取模型的输出张量（实际上，也可以是任何权重、偏置或特征张量），从而实现自定义分析操作。以下我们以测试 LeNet-5 的识别正确率为例，演示相关函数的调用方法。

第一步：通过`load_from_file`函数加载模型的网络结构和参数。

```
quant_layer_list, quant_param_dict = \
       ndk.modelpack.load_from_file(
              fname_prototxt='example_lenet5_quant.prototxt',
              fname_npz='example_lenet5_quant.npz'
              )
```

第二步：准备一个用以提供测试图像和标签的生成器。我们使用 MNIST 测试集。简单起见，本例让每一个 batch 都输出全部的 10000 张图像。

```
g_test = mnist.data_generator_mnist( mnist_dirname='mnist',
                                     batch_size=10000,
                                     random_order=False,
                                     use_test_set=True
                                     )
```

第三步：获取 10000 张测试图，并调用 `run_layers` 函数执行模型计算任务。

在配置目标张量名时，由于统计识别率只需要监控模型最终的输出张量，以下代码中我们直接通过调用 `get_network_output` 函数直接获取网络的输出。

```
data_batch = next(g_test)
output_tensor_name = ndk.layers.get_network_output(layer_list)

test_output = ndk.quant_tools.numpy_net.run_layers(
                  input_data_batch=data_batch['input'],
                  layer_list=quant_layer_list,
                  target_feature_tensor_list=output_tensor_name,
                  param_dict=quant_param_dict,
                  bitwidth=8,
                  quant=False,
                  log_on=True
                  )
```

第四步：统计识别正确率。

```
correct_prediction = \
        np.equal( np.argmax(test_output[output_tensor_name], 1),
                  np.argmax(data_batch['output'], 1)
                  )
accuracy = np.mean(correct_prediction)
print('Accuracy={:.3f}%'.format(accuracy*100))
```

以上代码是浮点模型的识别正确率进行统计，运行结果为：

```
Accuracy: 99.04%
```

对定点模型的识别正确率进行统计，只需在以上第三步中将 `run_layers` 函数的参数配置为 `quant=True`。运行结果为

```
Accuracy=99.01%
```

使用定点模型训练调整权重/偏置可以显著提高定点模型性能：

```
Accuracy=99.12%
```

注意到，在经过定点模型训练之后模型的识别正确率比原先浮点模型的更高。这是因为定点模型训练不是简单的定点化操作，实际上是对神经网络在定点参数限制下的重新训练。在原浮点模型性能存在优化空间的情况下，出现定点性能比浮点性能更好的情况是合理的。

9.6 导出模型为二进制文件

在取得了定点模型参数之后，接下去就要导出到芯片平台了。

通过简单地调用函数 modelpack_from_file 就可以实现。

```
ndk.modelpack.modelpack_from_file(
              bitwidth=8,
              fname_prototxt='example_lenet5_quant.prototxt',
              fname_npz='example_lenet5_quant.npz',
              out_file_path='out',
              model_name='lenet5'
              )
```

运行以上代码，在 out_file_path 指定的路径下会生成一个名称为"lenet5.bin"的文件。将这个二进制文件配合 SDK 使用，就可以在芯片平台上运行手写体识别的 LeNet-5 神经网络了:)

9.7 在 SDK 中调用

将生成的 lenet5.bin 拷贝到 SDK 应用程序工程代码路径下的 custom 文件夹里。

在编写代码时，用户可以通过以下示例代码加载模型，然后将存放在 `input_sample_0` 所指内存中的一个输入特征张量送进网络进行计算，计算结果会被转存到 `net_output` 所指向的地址。

```
struct cnn_net_pack_info pack_info = cnn_load_net_pack("lenet.bin");
int8_t *net_output = (int8_t *)os_mem_malloc(1, 10);
/* input is stored at addr input_sample_0 */
cnn_run_net_pack(&pack_info, input_sample_0, net_output); // run net
cnn_print_top5(net_output, 10);    // print result
cnn_delete_net_pack(&pack_info);  // delete net from memory
```