|  |  |  |  |
| --- | --- | --- | --- |
|  |  | |  |
|  | |  |
| **昇腾模型开发项目开发交付文档** | |
|  |  |
|  |  |

目 录

[1 模型概述 4](#_Toc60617904)

[1.1 模型来源 5](#_Toc60617905)

[1.1.1 生成模型 5](#_Toc60617906)

[1.1.2 数据拟合 6](#_Toc60617907)

[1.2 RealNVP概述 7](#_Toc60617908)

[1.2.1 Flow：流问题 7](#_Toc60617909)

[1.2.2 RealNVP的加性耦合层 8](#_Toc60617910)

[1.2.3 RealNVP的特征 9](#_Toc60617911)

[1.2.3.1 压缩处理 9](#_Toc60617912)

[1.2.3.2 多尺度架构 10](#_Toc60617913)

[2 开发设计方案 11](#_Toc60617914)

[2.1 迁移评估 12](#_Toc60617915)

[2.1.1 迁移方式评估 12](#_Toc60617916)

[2.1.2 迁移代码 12](#_Toc60617917)

[2.1.3 精度分析 14](#_Toc60617918)

[2.2 设计方案 15](#_Toc60617919)

[2.2.1 方案来源 15](#_Toc60617920)

[2.2.2 TensorFlow迁移过程 15](#_Toc60617921)

[3 交付结果 18](#_Toc60617922)

[3.1 模型网络文件 19](#_Toc60617923)

[3.2 模型训练脚本与训练超参 20](#_Toc60617924)

[3.3 训练、测试数据集链接 21](#_Toc60617925)

[3.4 训练过程 22](#_Toc60617926)

[3.5 模型转换 23](#_Toc60617927)

[3.5.1 checkpoint转换 23](#_Toc60617928)

[3.5.2 pb生成 24](#_Toc60617929)

[3.5.3 om转换 25](#_Toc60617930)

[3.6 是否同意官方托管相关模型 27](#_Toc60617931)

[4 精度性能数据 29](#_Toc60617932)

[4.1 训练精度 30](#_Toc60617933)

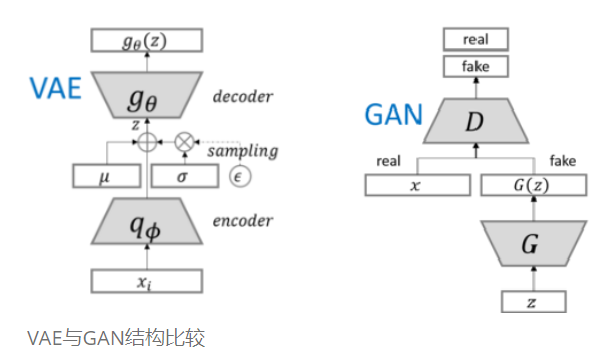
[4.2 训练性能 31](#_Toc60617934)

# 模型概述

## 模型来源

### 生成模型

目前主流的生成模型有VAE和GAN，但除了这两个之外，还有基于 flow 的模型。flow 可以直接翻译为“流”，从模型的角度可以很直观的得到这一结论。



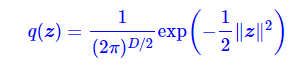
生成模型的本质，就是希望用一个我们知道的概率模型来拟合所给的数据样本，也就是说，我们得写出一个带参数的分布。然而，我们的神经网络只是“万能函数拟合器”，却不是“万能分布拟合器”，也就是它原则上能拟合任意函数，但不能随意拟合一个概率分布，因为概率分布有“非负”和“归一化”的要求。这样一来，我们能直接写出来的只有离散型的分布，或者是连续型的高斯分布。

从数学的角度来看，图像应该是一个离散的分布，因为它是由有限个像素组成的，而每个像素的取值也是离散的、有限的，因此可以通过离散分布来描述。这个思路的成果就是 PixelRNN 一类的模型了，我们称之为“自回归流”，其特点就是无法并行，所以计算量特别大。所以，我们更希望用连续分布来描述图像。当然，图像只是一个场景，其他场景下我们也有很多连续型的数据，所以连续型的分布的研究是很有必要的。

### 数据拟合

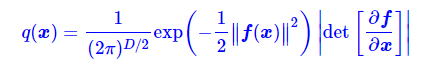
对于连续型的数据，可以写出高斯分布的表达公式。通常为了方便处理，我们只能写出各分量独立的高斯分布。考虑到高斯分布只是众多连续分布中极小的一部分，我们通过积分来创造更多的分布：

其中，各公式含义如下：

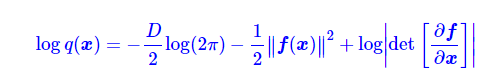


理论上来看，该分布函数能拟合任意分布。为了求出参数θ，采用最大似然法。假设真实数据分布为 p̃(x)，我们有：

考虑到该变换是在概率密度函数上进行，即不是的简单代换，因此在替换过程中将产生雅可比行列式。具体的来看，可以看到的解析形式如下：



进一步的，考虑其最大似然可知：



其中，f是转换函数g的逆函数，用于求解。由上述表达可以看出，如果希望求解的值，则对于f有以下要求：

1. 该函数是可逆的。通过可逆函数，我们可以求得原始函数g，即为期待生成的概率模型。
2. 该函数的雅可比行列式较容易计算，即的最后一项求解较为容易。从模型的公式上来看，前两项均可在多项式时间内计算得到，而求解行列式所需要的时间较多，因此我们需要在这一项上进行限制。

## RealNVP概述

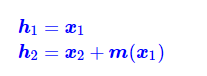
以下介绍RealNVP模型，该模型是对Flow模型的改进。以下我们将从原函数g和逆函数f的角度分析RealNVP模型中，概率分布拟合的特征与方式。在下文表述中，我们将逐步从数学公式转换到论文中提出的拟合过程，因此在变量命名中有一定的差异。这部分将在本文的论述中给出详细的对应关系，方便读者阅读。

### Flow：流问题

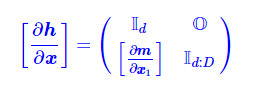
考虑以下可能的逆函数f（即Flow模型论文中的公式x）：



其原函数g（即Flow模型论文中的公式中的h）为以下形式：



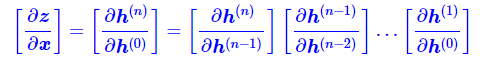
对于这样一个模型，其雅可比行列式（即上节中的）为：

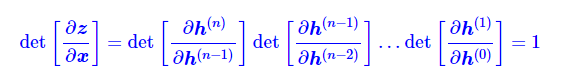


可以看到，这样的雅可比行列式的值是很容易求得的，它的值为对角线上的元素1，其对数为 0，很好的解决了行列式的计算问题。同时，该函数的逆函数非常容易得到。因此，为了使这一变换达到很强的非线性，我们对这一变换进行复合，即：



由此，其雅可比行列式和行列式的值分别如下：

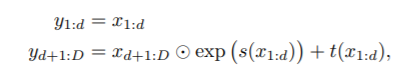




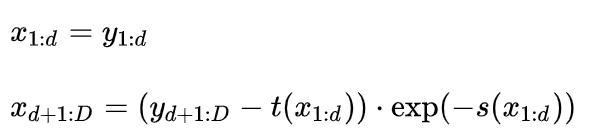
由此，我们得到了一个Flow模型。我们将这种变换h和x记为加性耦合层，即Coupling Layer。

### RealNVP的加性耦合层

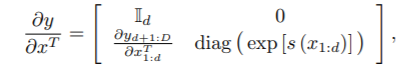
RealNVP是对Flow模型的改进，其中最重要的部分是对加性耦合层的缩放。考虑到在上述雅克比行列式中，只要上三角中不出现非零元素，那么该行列式就较为容易计算。因此，该模型提出了以下的转换函数g：

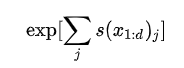


其逆函数f可以写成：

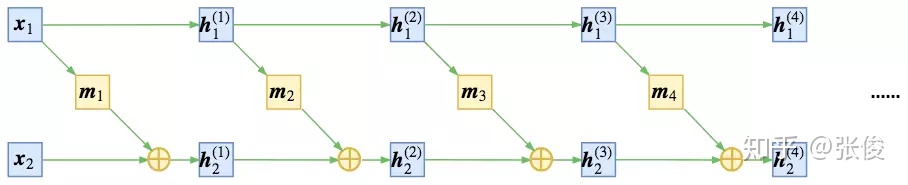


由此我们可以得到雅克比行列式为：

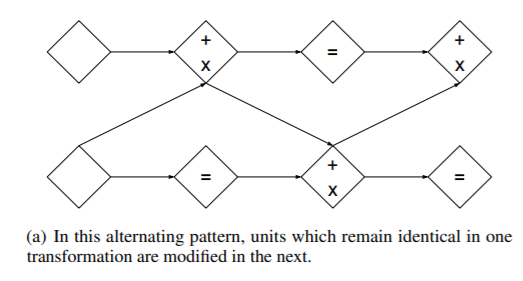


注意到该行列式的值也是对角线元素的乘积。因此，该模型同样可逆并且容易计算。

考虑到耦合的顺序如果一直是单向的，则不会产生非线性的性质（事实上，这种情况仅仅是对模型的一种线性组合）。



因此，我们需要将模型的流顺序进行调换，以增强其非线性的性质。

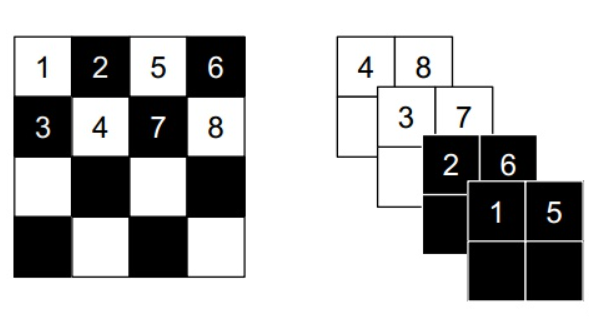


以上就是对RealNVP模型中加性耦合层的描述。

### RealNVP的特征

以下我们将对RealNVP中的压缩处理和多尺度架构进行解释，这些性质对应于实现中的部分代码，故在本节中进行列举。从中也可以看到RealNVP在流模型领域的改进。

#### 压缩处理



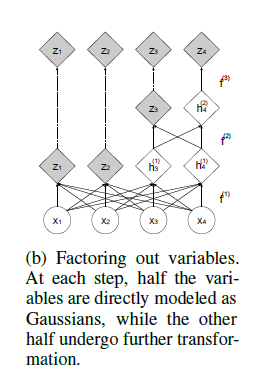
假设原通道数为c，尺寸为，则以为块切分区域，并把块区域沿通道方向平铺开来，形成尺寸为，通道数为h\*h\*c的张量。该操作为压缩处理，压缩了图像的尺寸，并且加深了通道数。在保持相关性的情况，同时起到打散变量的作用。如下图像，切分块的h为2。

#### 多尺度架构

多尺度架构实际上是当经过一层的卷积层流处理后，一半的通道直接作用于到输出层，而另一半的通道再去经过流操作，每次都有一半的通道做更复杂的变换。这种多尺度架构有效的降低了模型复杂度，引入了正则的效果。

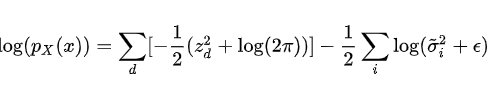
整个过程描述如下：

1. 输入x通过卷积耦合层，得到
2. 取 到最终的输出 ,而另一半重复步骤1，过卷积耦合层后，在取一半做最终输出，而另一半重复步骤1，堆叠n层。



对于最终的连接耦合层，由于各通道变量实际上独立，故最终的行列式为各通道最终输出行列式的连乘。

在最后，输出层经过batch norm处理，即有。输出的雅可比行列式为。一般变换到高斯分布空间，最终的损失函数为：



利用逆变换传播误差，反向训练非线性的卷积网络s和t。

# 开发设计方案

包括并不限于：

* 评估分析模型训练迁移难易程度、工作量、开源代码&数据集可获得性等
* 基于开源代码和数据集跑通CPU/GPU训练，并确认训练迁移精度目标
* 训练迁移设计方案（适配点）
* 模型转换及推理设计方案

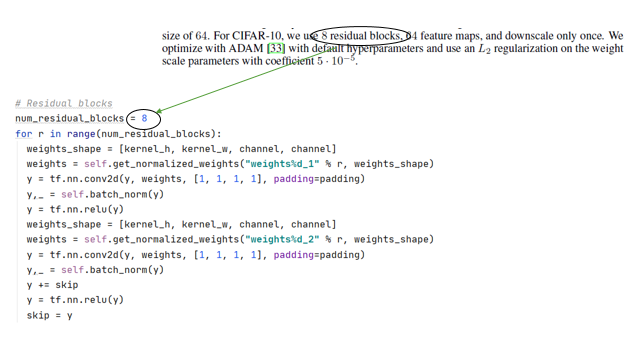
## 迁移评估

### 迁移方式评估

模型迁移的常规方法是通过开源社区平台进行迁移。具体的来看，可以考虑通过Google TensorFlow框架、PyTorch转换和基于平台模型进行修改三种方式获得。在TensorFlow框架中已经具备了该模型的代码。但是TensorFlow中仅实现了模型用于拟合数据，没有给出很好的方式去验证数据的正确性。考虑到loss函数定义较为模糊，加上本次模型迁移的数据集是cifar-10数据集，因此采用第三种方式进行实现。

### 迁移代码

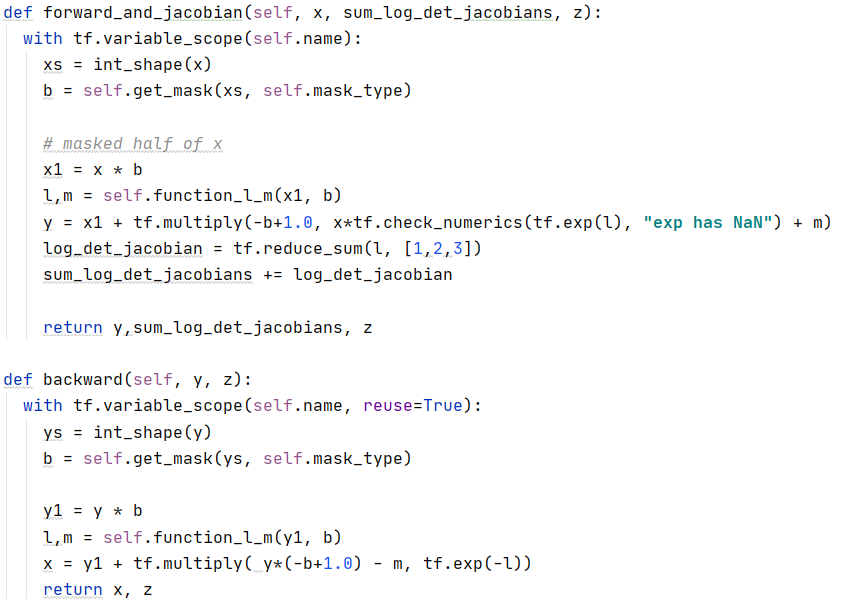
迁移代码主要通过GitHub进行获取，在迁移代码中，主要是针对论文中提到的相关参数进行优化。例如根据论文中的要求，采用8个残差块、64个feature map急性优化，在Adam优化其中采用默认参数进行实现等。



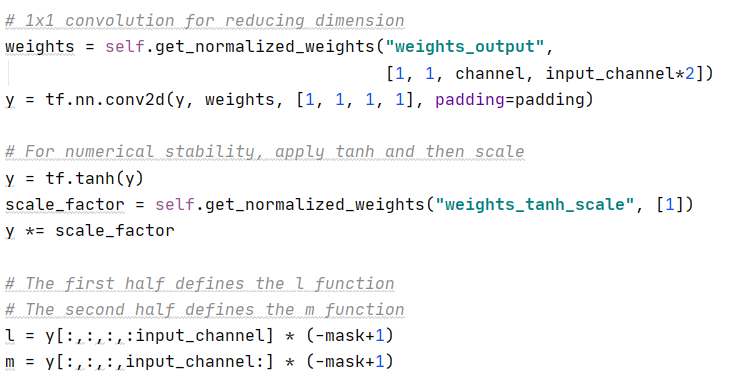
在Optimizer部分，自实现往往会导致不收敛的情况。因此，我们直接对loss函数进行优化，采用learning rate超参进行控制。具体代码如下：



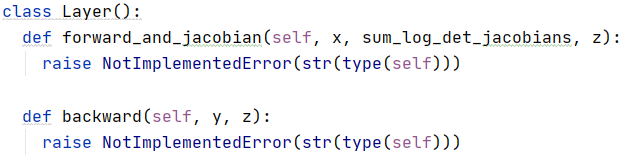
在Coupling Layer实现部分，采用前向和后向传播方式进行编写。可以看到这一部分与钱问题到的RealNVP模型中的加性耦合层的函数完全一致。



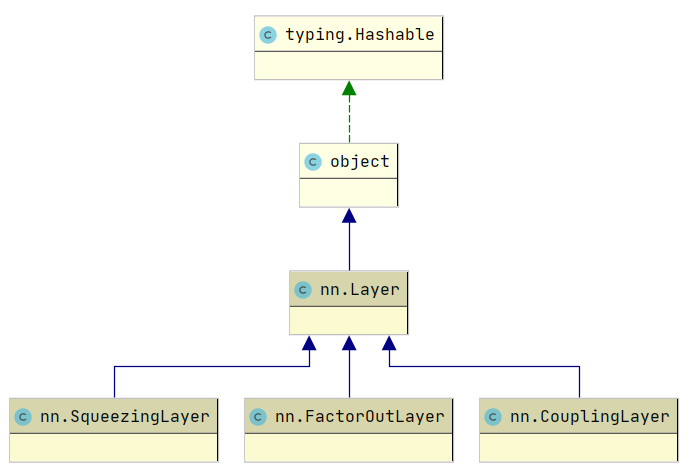
其中，function\_l\_m的定义包括了残差块的定义（前文已提及）和如下部分（池化层与激活函数），主要目的在于实现一个可调参数的神经网络，用于对分布进行拟合。



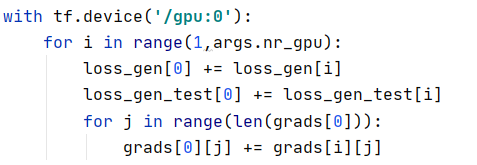
在迁移过程中，为了确保模型之间的低耦合性。我们考虑以接口的方式进行实现。具体的来看，我们定义了接口层，用于实现相关的接口调用，其代码如下。



在这一接口实现后，Copling Layer、Squeezing Layer、FactorOutLayer均继承于这一接口进行实现。考虑到模型的建构过程中仅使用接口中提到的前向和后向传播函数，而忽略其它辅助函数，因此在这一过程中模型的可重用性更高，其类图关系如下：



最后，考虑训练过程是基于NPU进行的，GPU的分布式训练在NPU中很难完成，因此我们需要对分布式训练进行优化。具体的，我们将nr\_gpu指定为1，将模型放入昇腾NPU环境中。代码如下：



在超参部分，调整相关参数以匹配华为模型迁移平台。



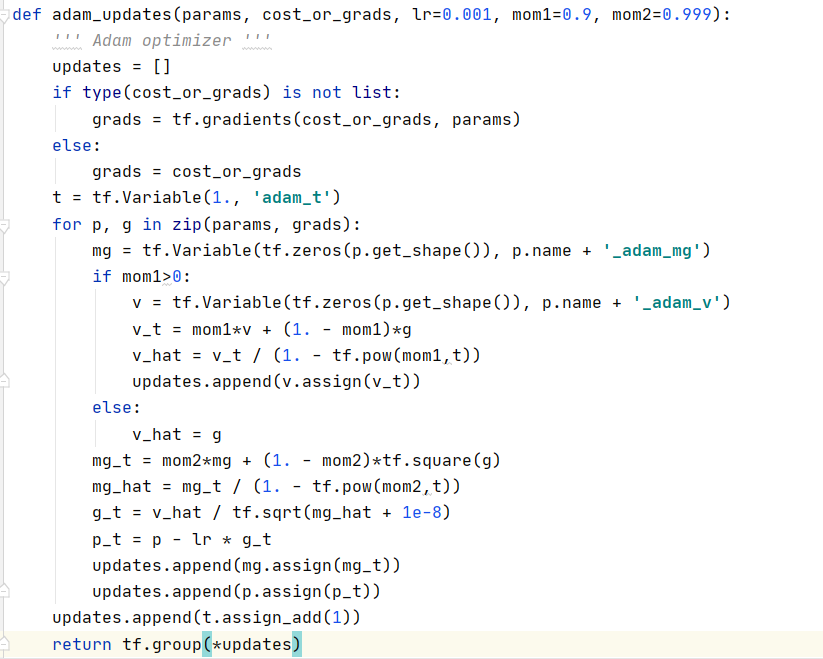
### 精度分析

代码实际是将原始代码进行改动，从而获得真实的可运行版本。在获取代码的过程中，我们采用了GitHub第三方代码进行实现。在今后的版本中，也可以尝试基于Google TensorFlow的版本进行实现，但是需要指定bits per dimension的定义。在此版本代码中，loss的定义如下：



实际上，RealNVP文章中没有这一部分的定义，该定义是参考Flow模型得到的，关于Flow模型的介绍参考前文）。

另外，Adam Optimizer的定义可参考以下代码。实际上，TensorFlow本身就有相关的实现，且昇腾环境下支持这一算子的运行，因此采用TF框架下的代码是一个更好的选择。



最后，基于该开源代码和数据集，我们可以在 CPU上进行运行，一次5小时左右的时间。可参考以下log获取CPU运行时间的参考文档。代码在CPU中的运行时间大约是5小时/epoch。根据论文中的描述，考虑到论文中提到的精度是最佳实现精度。而文章本身没有给出详细的代码实现，因此经过分析，初步将模型的目标定为能够达到3.9的训练精度。

## 设计方案

### 方案来源

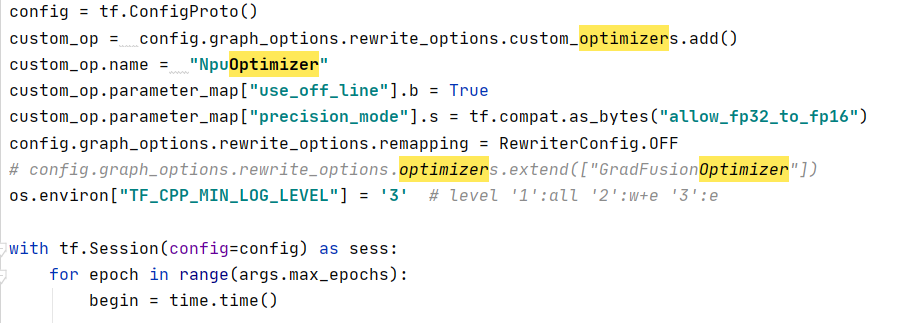
根据昇腾在TensorFlow上的迁移指南，我们可以据此完成模型的迁移工作。主要的适配点是采用npu\_bridge的包进行导入。

### TensorFlow迁移过程

根据流程，首先我们使用sess.run进行迁移。具体的，需要将以下内容导入npu\_bridge中。

from npu\_bridge.estimator import npu\_ops   
from npu\_bridge.estimator.npu\_unary\_ops import npu\_unary\_ops  
from npu\_bridge.estimator.npu.npu\_config import NPURunConfig  
from npu\_bridge.estimator.npu.npu\_estimator import NPUEstimator,NPUEstimatorSpec  
from npu\_bridge.estimator.npu.npu\_optimizer import NPUDistributedOptimizer  
from npu\_bridge.estimator.npu import npu\_loss\_scale\_optimizer  
from npu\_bridge.estimator.npu import npu\_loss\_scale\_manager

同时，考虑以下的配置更新，用于配置离线推理的相关内容。



具体的，可参考以下部分解释各配置的含义。

|  |  |
| --- | --- |
| 配置项 | 说明 |
| use\_off\_line | 是否在昇腾AI处理器执行训练。   * True：在昇腾AI处理器执行训练。 * False：在Host侧的CPU执行训练，默认为False。 |
| profiling\_mode | 是否开启Profiling功能。   * True：开启Profiling功能，从enable\_options读取Profiling的采集选项。 * False：关闭Profiling功能，默认关闭。 |
| precision\_mode | 算子精度模式，配置要求为string类型。   * allow\_fp32\_to\_fp16或None：算子不支持float32类型时，直接选择float16。默认选择此种模式。 * force\_fp16：算子既支持float16又支持float32数据类型时，强制选择float16。 * must\_keep\_origin\_dtype：保持原图精度。 * allow\_mix\_precision：开启自动混合精度功能，针对全网中float32数据类型的算子，系统会按照内置优化策略自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。开启该功能开关后，用户可以同时使能Loss Scaling，从而补偿降低精度带来的精度损失，相关接口请参考NPULossScaleOptimizer构造函数。 |
| mix\_compile\_mode | 是否开启混合计算模式。   * True：开启。 * False：关闭，默认关闭。   计算全下沉模式即所有的计算类算子全部在Device侧执行，混合计算模式作为计算全下沉模式的补充，将部分不可离线编译下沉执行的算子留在前端框架中在线执行，提升昇腾AI处理器支持Tensorflow的适配灵活性。 |
| graph\_run\_mode | 图执行模式，取值：   * 0：在线推理场景下，请配置为0。 * 1：训练场景下，请配置为1，默认为1。 |

# 交付结果

## 模型网络文件

使用train.py进行训练。相关层的定义在nn.py中给出。相关的requirements列举如下，可根据以下配置进行Python环境设置，并执行相关代码。Util.py用于生成RealNVP的样例，从代码执行的角度可以不考虑该文件。

|  |  |  |
| --- | --- | --- |
| absl-py==0.10.0 | astor==0.8.1 | astunparse==1.6.3 |
| bleach==1.5.0 | cachetools==4.1.1 | certifi==2020.6.20 |
| chardet==3.0.4 | cloudpickle==1.6.0 | cycler==0.10.0 |
| decorator==4.4.2 | dm-tree==0.1.5 | enum34==1.1.10 |
| gast==0.2.2 | google-auth==1.22.1 | google-auth-oauthlib==0.4.1 |
| google-pasta==0.2.0 | grpcio==1.32.0 | h5py==2.10.0 |
| html5lib==0.9999999 | idna==2.10 | importlib-metadata==2.0.0 |
| Keras-Applications==1.0.8 | Keras-Preprocessing==1.1.2 | kiwisolver==1.2.0 |
| Markdown==3.3.1 | matplotlib==3.3.2 | mock==4.0.2 |
| numpy==1.18.5 | oauthlib==3.1.0 | opt-einsum==3.3.0 |
| Pillow==8.0.0 | protobuf==3.13.0 | pyasn1==0.4.8 |
| pyasn1-modules==0.2.8 | pyparsing==2.4.7 | python-dateutil==2.8.1 |
| requests==2.24.0 | requests-oauthlib==1.3.0 | rsa==4.6 |
| scipy==1.4.1 | six==1.15.0 | tensorboard==1.13.1 |
| tensorboard-plugin-wit==1.7.0 | tensorflow==1.13.1 | tensorflow-estimator==1.13.0 |
| tensorflow-probability==0.11.1 | tensorflow-tensorboard==0.4.0 | termcolor==1.1.0 |
| urllib3==1.25.10 | Werkzeug==1.0.1 | wrapt==1.12.1 |
| zipp==3.3.0 |  |  |

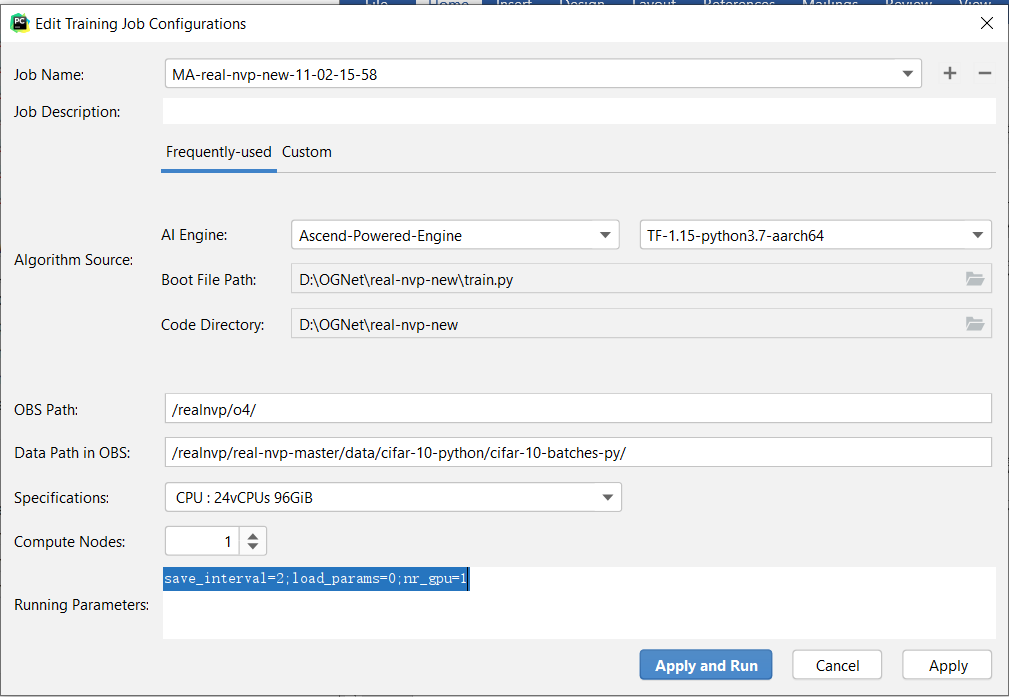
具体的，可以参考附件中Pull Request的代码。值得注意的是，实际运行中可能不需要一部分文件，但这些文件依然被包括在上传的代码中。原因是在于使用这些文件，你可以很容易的得到一些模型迭代的部分结果。这些结果有助于检查代码中存在的部分问题。

## 模型训练脚本与训练超参

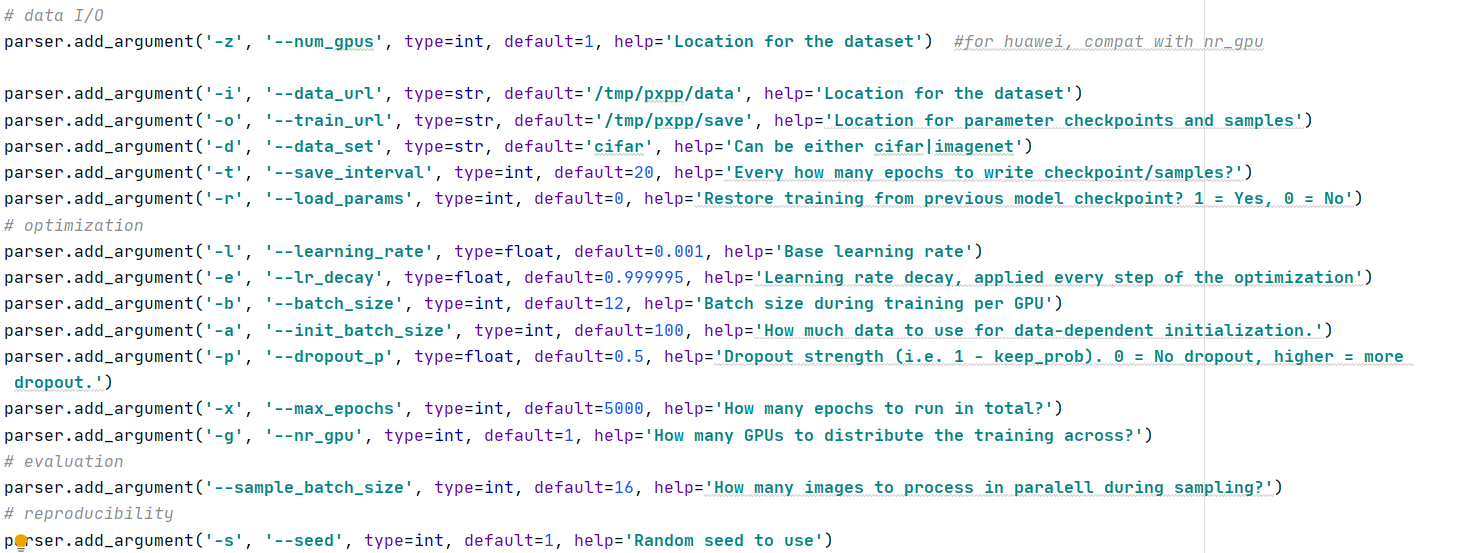
模型训练脚本参考附录即可。具体的，可以考虑以下最佳实践（该参数可以在PyCharm Toolkit环境下成功运行），其余未提及参数均采用默认形式。

save\_interval=2;load\_params=0;nr\_gpu=1

以下是在PyCharm Toolkit中的其它参数，包括了训练任务的AI Engine等配置，需要将模型手工转换到Ascend平台和TF框架进行运行。

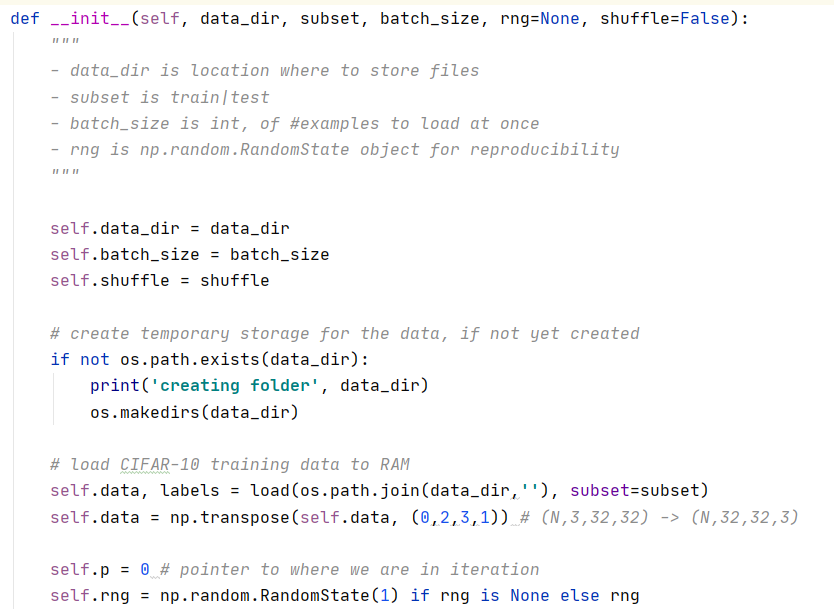


以下是对默认参数的解释，可以尝试对这些参数进行微调，以适配不同的环境。特别注意的是-l和-e参数，这两个参数决定了学习率，可以动态调节。其次，-b指定了batch size的大小。其余参数均用于默认的配置以及IO读取。Num\_gpus用于适配华为昇腾环境中提到的必选参数。



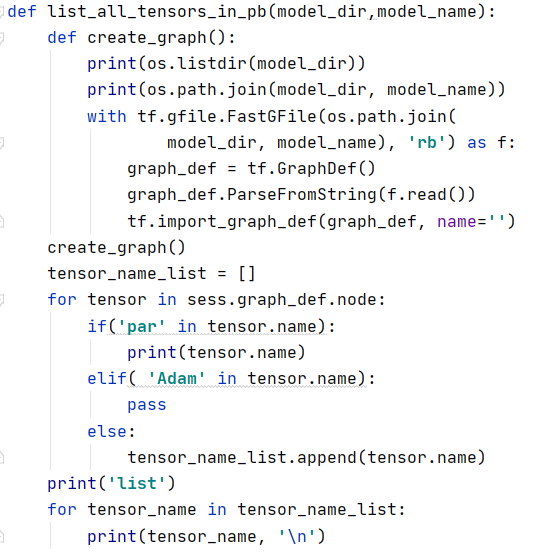
## 训练、测试数据集链接

根据模型要求，数据集使用cifar-10数据集。对数据集的解析可以参考cifar-data.py文件。其中，该模型不需要使用label标签，故在进行模型解析时删除了这一部分。具体代码如下：

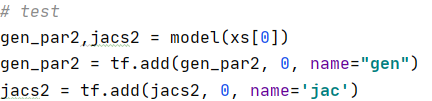


## 训练过程

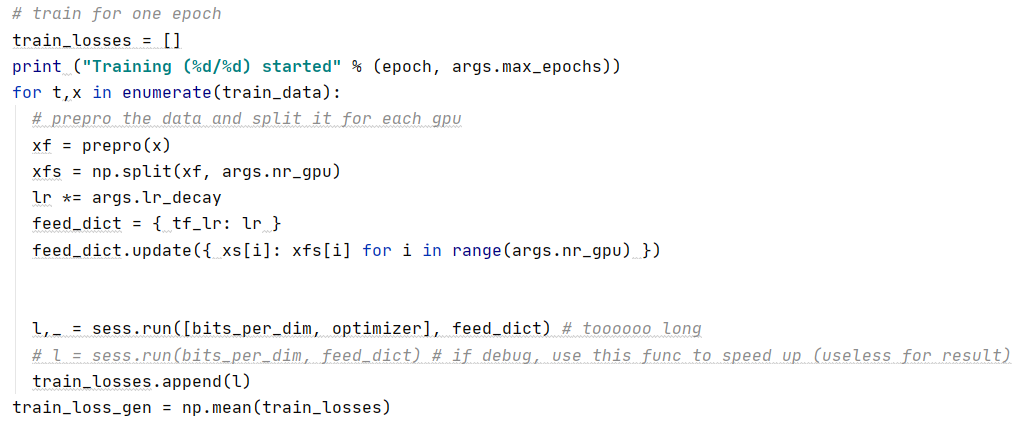
训练过程参考正常的训练过程即可，即运行train.py。为了便于调试，特别是在模型导出期间的调试，我们使用以下函数列出模型中所具备的所有tensor。同时，我们重命名了输出tensor用于在模型输出中便于查找。



模型输出主要考虑inference部分，修改代码如下：



具体的训练过程如下，基本思路是进行数据预处理，然后进入模型中进行运行，采用相应的优化器进行优化。



为了方便debug，可以考虑注释optimizer以节约时间。

## 模型转换

### checkpoint转换

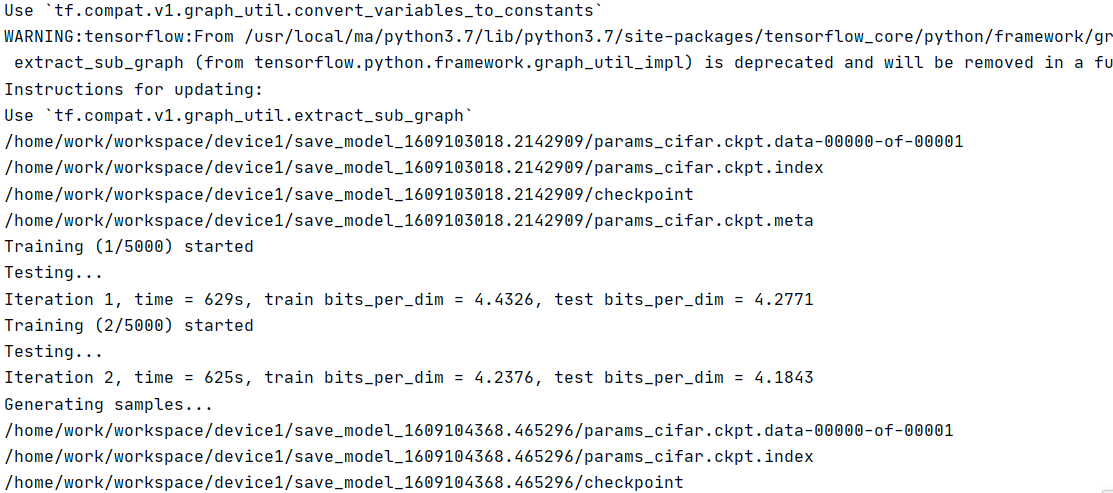
在这一部分中，主要实现了checkpoint的转换。使用S3 compatible接口可以很容易的做到这点。同时，华为的moxing包也可以支持相同效果。具体的代码如下：



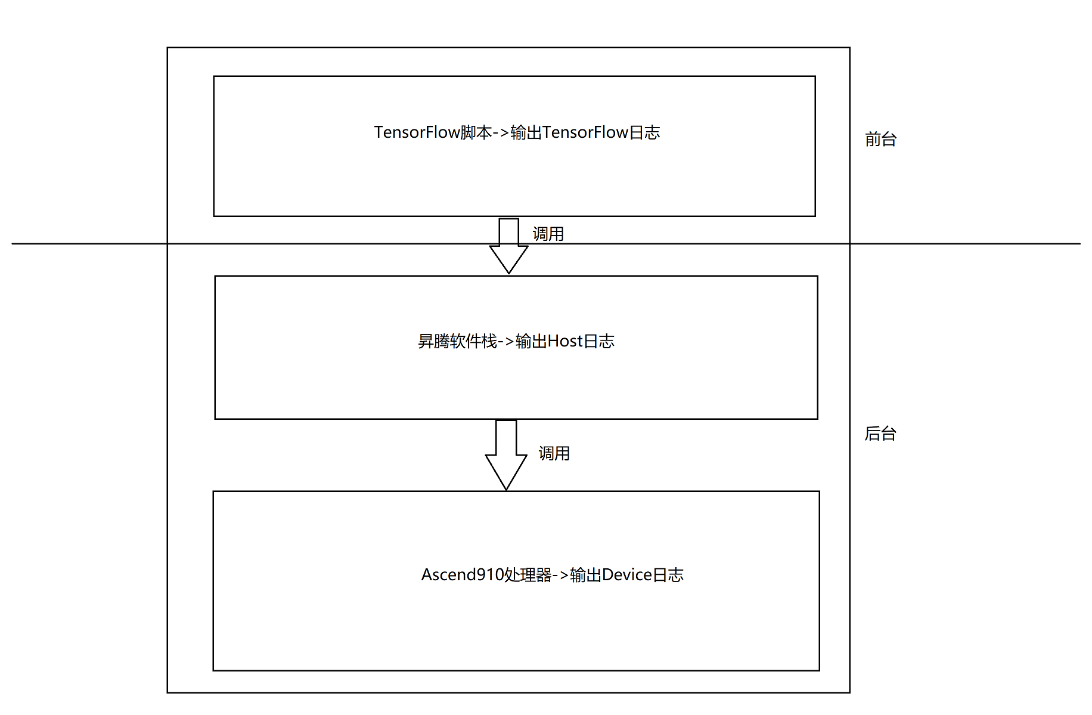
注意到boto3是S3接口的Python实现，在不同语言中有不同的S3接口调用函数，可以基于此进行实现和修改。

### log输出

Log的输出在模型中有十分重要的位置，在log中插入关键字可以有效地降低debug的时间，具体的可以参考tensorflow的日志级别以及自定义的Iteration和文件名log区别。根据这一log可以很快定位最佳loss下的ckpt的位置，方便后续的生成。



具体的来看，日志可分为以下几种类型，分别称为TensorFlow（或者叫前台）日志，Host日志，Device日志，他们之间的关系如下：



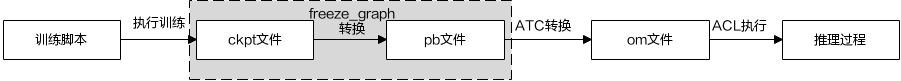
TensorFlow（前台）日志是在启动训练之后，在日志打印窗口能直接看到的日志输出。Host日志在定位问题的角度，作用是比较大的。特别的，在提交了ISSUE之后会涉及这部分内容。可以通过代码获取这部分日志。

os.environ['SLOG\_PRINT\_TO\_STDOUT'] = "1"

Device日志是执行具体任务的硬件侧打印的，某些时候这部分日志包含的信息很关键，这部分日志是不可见的。

### pb生成

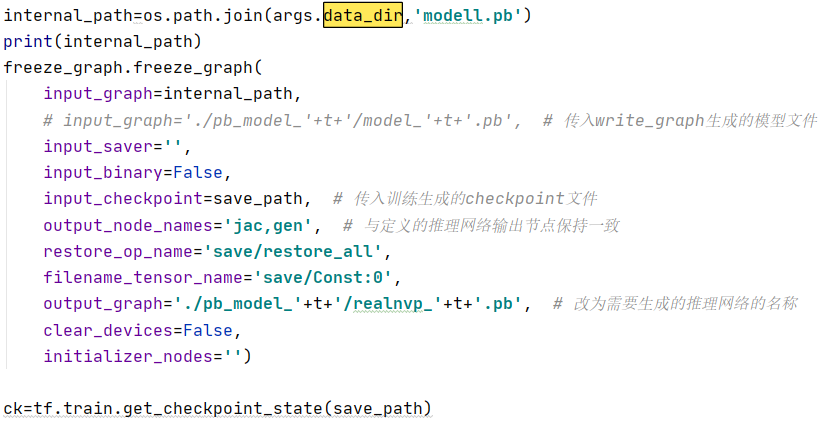
tensorflow在训练过程中，通常使用saver = tf.train.Saver()和saver.save()保存模型，一次saver.save()后会生成ckpt文件。这种模型权重数据和模型结构是分开保存的方式，在推理场景下，一般使用tensorflow提供的freeze\_graph函数，将权重数据和模型结构合并为pb格式的文件，对应下图虚框所示部分。



具体的，从代码实现的角度，我们首先需要生成model.pb文件用于保存模型结构（仅名称，非数据）。代码如下：

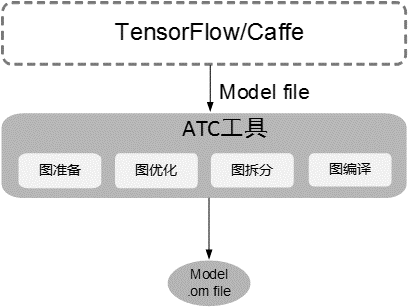
tf.train.write\_graph(sess.graph\_def, './pb\_model', 'model.pb')

接下来，我们需要在这个文件中加载ckpt中保存的值，代码如下：



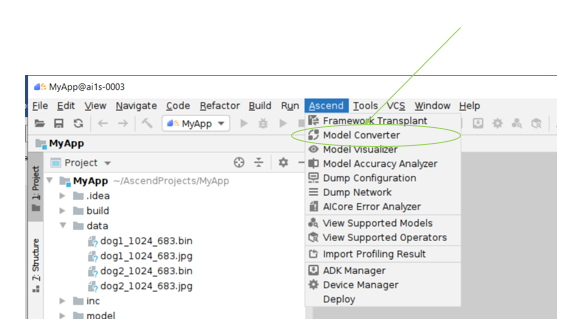
### om转换

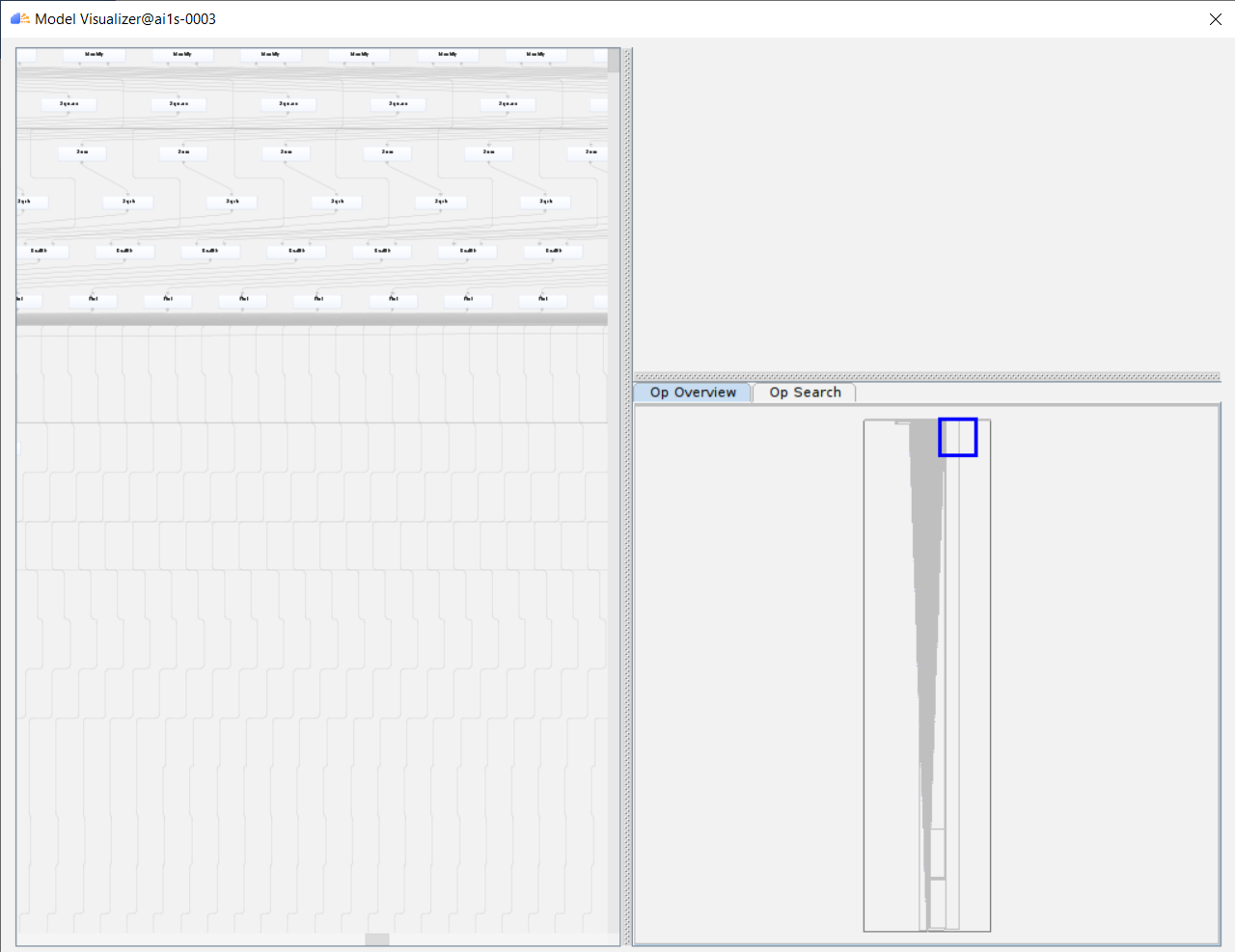
我们已经训练好了一个Tensorflow的模型，接下来我们需要通过ATC工具将其转换为昇腾AI处理器支持的离线模型，模型转换过程中可以实现算子调度的优化、权重数据重排、内存使用优化等，可以脱离设备完成模型的预处理，详细架构如图所示。

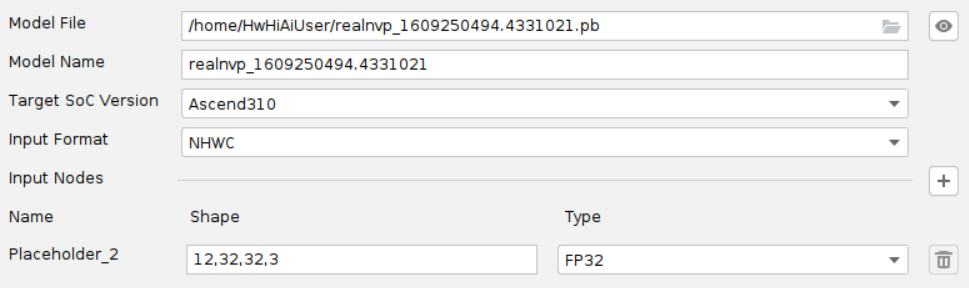


在Mind Studio服务器上使用MindStudio-ubuntu/bin/MindStudio.sh文件以启动ATC转换工具。

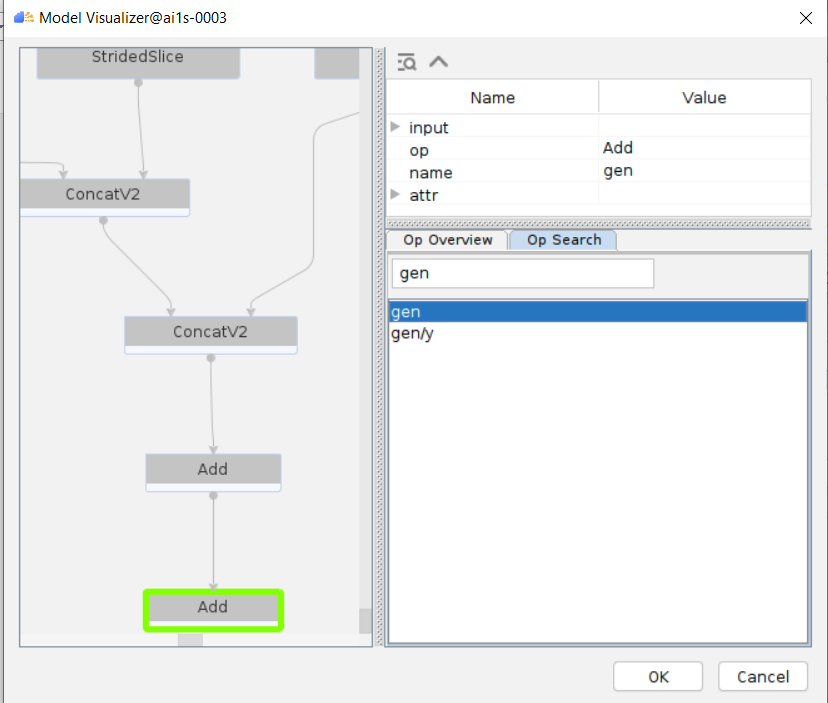
进入后可以看到Model Converter，该工具可用于模型转换。

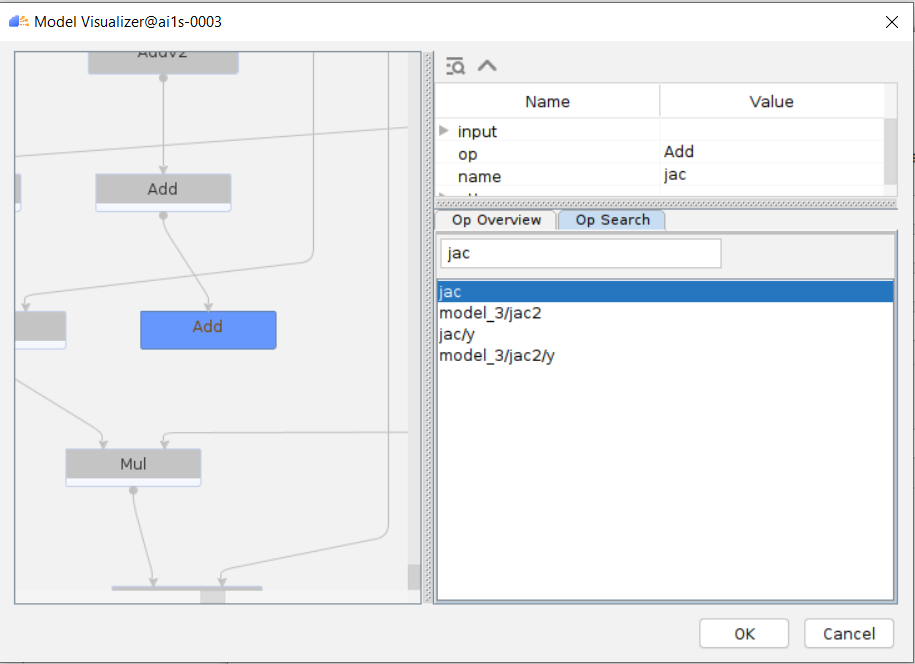


以下我们将模型引入，可以看到相应的模型图生成。



考虑到节点较多，我们将指定的输出节点（该工具提供了查找功能）在该模型图中标出，即可进行om文件生成。





最后，我们通过运行ATC工具即可得到相关的运行结果。

## 是否同意官方托管相关模型

是否同意官方托管相关模型：同意

allow-cache-pb/pth/om: True

allow-cache-pth/ckpt/pb/om

# 精度性能数据

## 训练精度

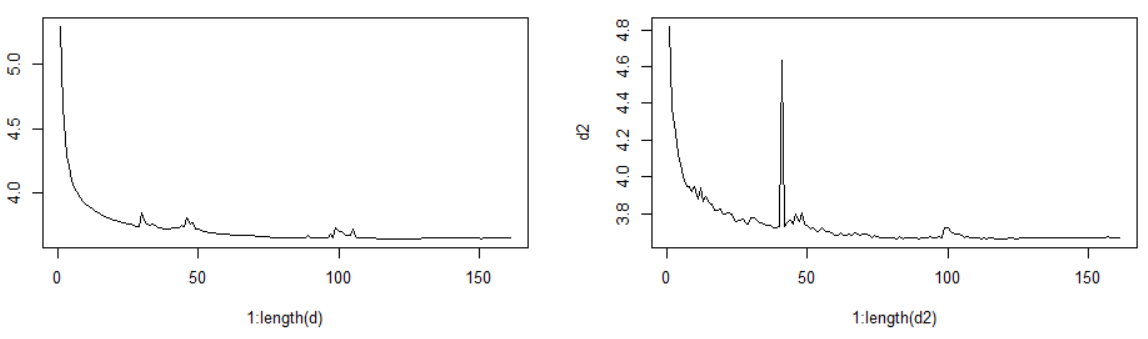
通过执行train.py --save\_interval=2 --load\_params=0 --nr\_gpu=1命令的log数据，得到训练的精度数据：

| Network | Network Type | Framework | Accuracy | Training Time | Precision | Dataset |
| --- | --- | --- | --- | --- | --- | --- |
| RealNVP | Flow | TensorFlow | 3.9 | 1000min | 3.65 | Cifar-10 |

其中，精度是在逐渐收敛的。以下我们展示不同训练次数下的精度数据与用时。

|  |  |  |
| --- | --- | --- |
| Time(min) | Train bpd | Test bpd |
| 837 | 5.2927 | 4.8197 |
| 7166 | 3.8992 | 3.8813 |
| 13617 | 3.7905 | 3.7945 |
| 20102 | 3.7751 | 3.7758 |
| 26636 | 3.7323 | 4.6336 |
| 33238 | 3.7156 | 3.7161 |
| 39927 | 3.6822 | 3.6853 |
| 46684 | 3.6712 | 3.6898 |
| 53514 | 3.6567 | 3.6678 |
| 60465 | 3.6553 | 3.6676 |
| 67530 | 3.7023 | 3.7052 |
| 74647 | 3.6539 | 3.6674 |
| 81780 | 3.6491 | 3.6646 |
| 88998 | 3.6531 | 3.6658 |
| 96256 | 3.6544 | 3.6703 |
| 103556 | 3.6522 | 3.6662 |
| 110952 | 3.6547 | 3.6705 |

进一步的，可以得到以下loss曲线。



由于昇腾环境下的限制（代金券等），该模型预期训练轮次需要5000轮。因此，我们有理由相信精度是可达标的。目前的环境下精度的数据维持在3.65左右。

## 训练性能

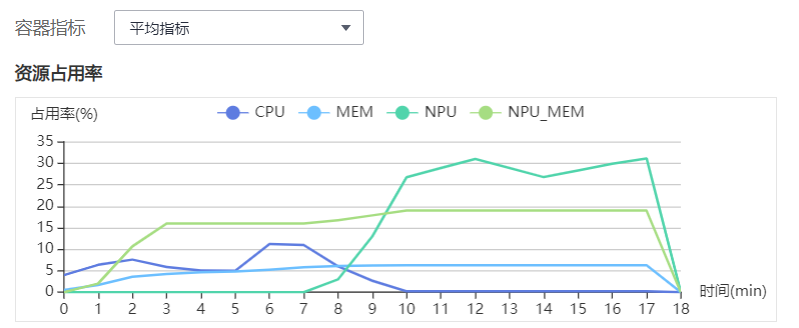
通过执行train.py --save\_interval=2 --load\_params=0 --nr\_gpu=1命令的log数据，得到训练的精度数据：

| Network | Network Type | Framework | Num of NPUs | Throughput | Batch Size | Precision | Dataset | Ascend AI Processor | NPU Version |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RealNVP | Flow | TensorFlow | 1 | 30% | 12 | Mixed | CIFAR-10 | Ascend 310 | Atlas300-9000-96GB |

以下我们报告相关的性能曲线。

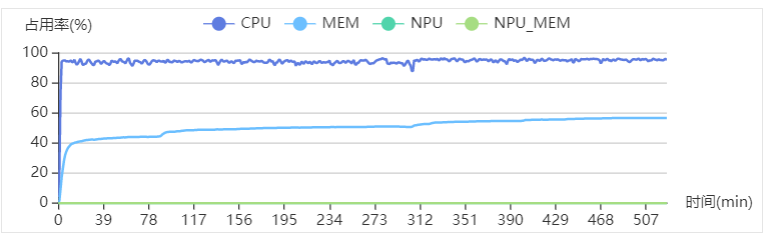
* 1. 数据预处理

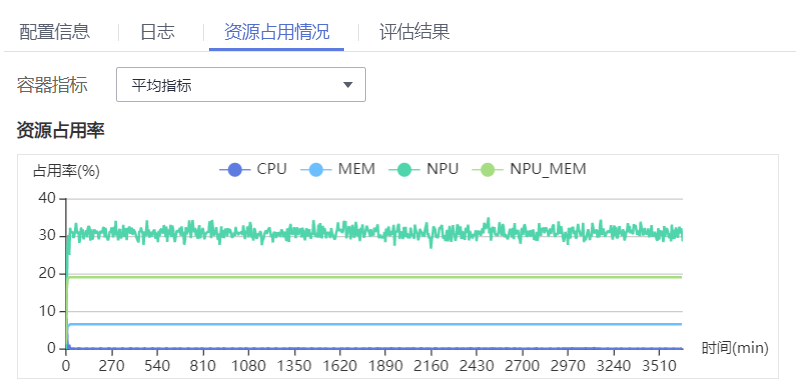
在数据预处理的过程中，大多数采用CPU进行预处理工作。随着昇腾平台的不断完善，这部分数据预处理的工作也可以在NPU上完成，从而更好的发挥昇腾平台的作用和优势。



* 1. CPU运行时间与NPU运行时间比较

昇腾平台的特性在这一步骤中可以得到体现。具体的可以看到，在使用CPU状态下，模型的单次epoch时间较长且CPU占用率很高。在NPU环境中，训练时间从5小时降低为10分钟，其优越性在该图表中得到了充分体现。





由于Tensorflow是采用npu bridge的方式在NPU侧运行。再加上模型的串行性特征，NPU的占用率不能达到100%的利用。这也与NPU对算子的优化有关，但是模型在CPU侧运行的部分几乎已经完全迁移到了NPU上，因此我们相信NPU在模型训练上的优势。