方案设计：Pytorch开源方案

一、特性列表

1、背景

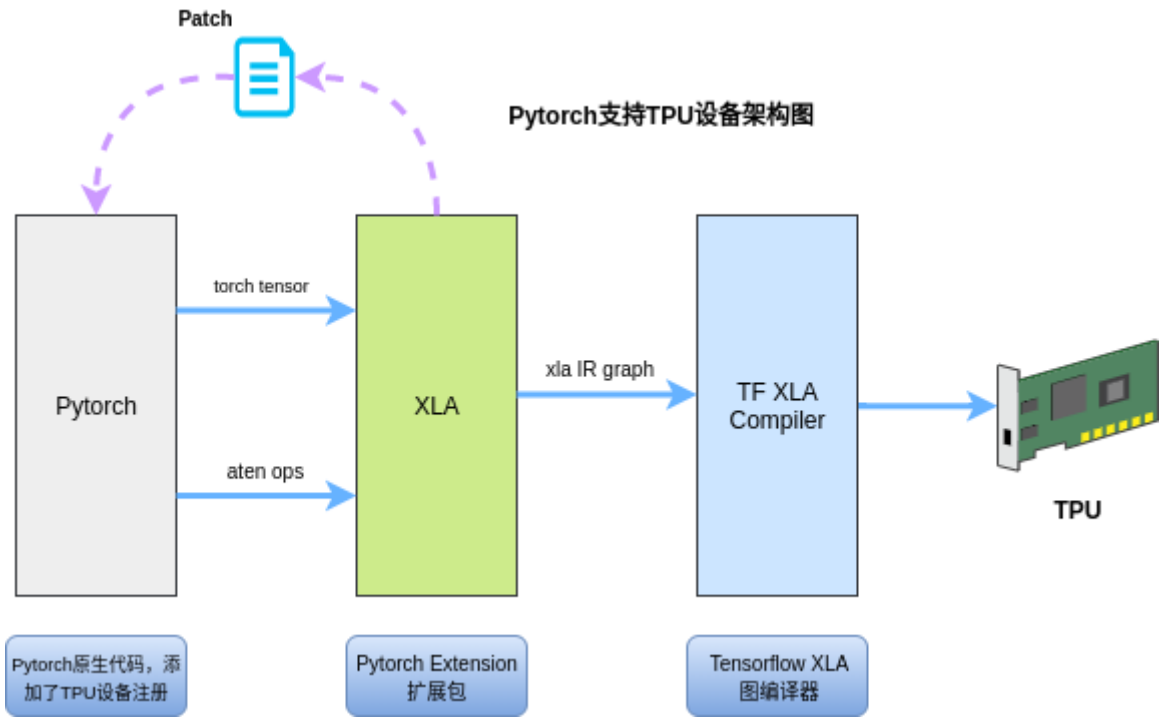
Pytorch作为一个新的深度学习框架，在业界越来越流行。开源Pytorch目前可支持CPU、GPU、TPU等设备上的推理和训练过程，Cambricon在开源Pytorch的基础上添加了MLU设备的支持。为了让更多的用户能够体验到MLU加速神经网络训练和推理的过程，计划对Cambricon Pytorch源码进行开源。

2、特性拆解

- 基于pytorch 1.6版本进行开源，所开源的代码和文档需要达到开源的标准；
- 开源的Catch代码能够做到训推一体;
- 开源的Catch代码接口上需要尽可能兼容原生pytorch的接口，减少用户迁移工作量;
- 开源的Catch代码能够进行可持续性开发和维护;
- 专利撰写;

二、竞品分析

Catch采用原生Pytorch extension扩展机制完成对MLU设备的支持，在目前已查到的开源软件中只有Google XLA采用这种方式对TPU设备进行支持，Google XLA具体的开源软件架构具体如下图所示：



在先前的版本中，Google XLA是以打patch的方式对原生Pytorch的代码进行修改，以便添加对TPU设备的支持。但较新的几个版本中，社区已经将XLA的patch合入到Pytorch主分支中，并随Pytorch版本进行发布和支持。

- Google XLA所支持的主要特性如下：

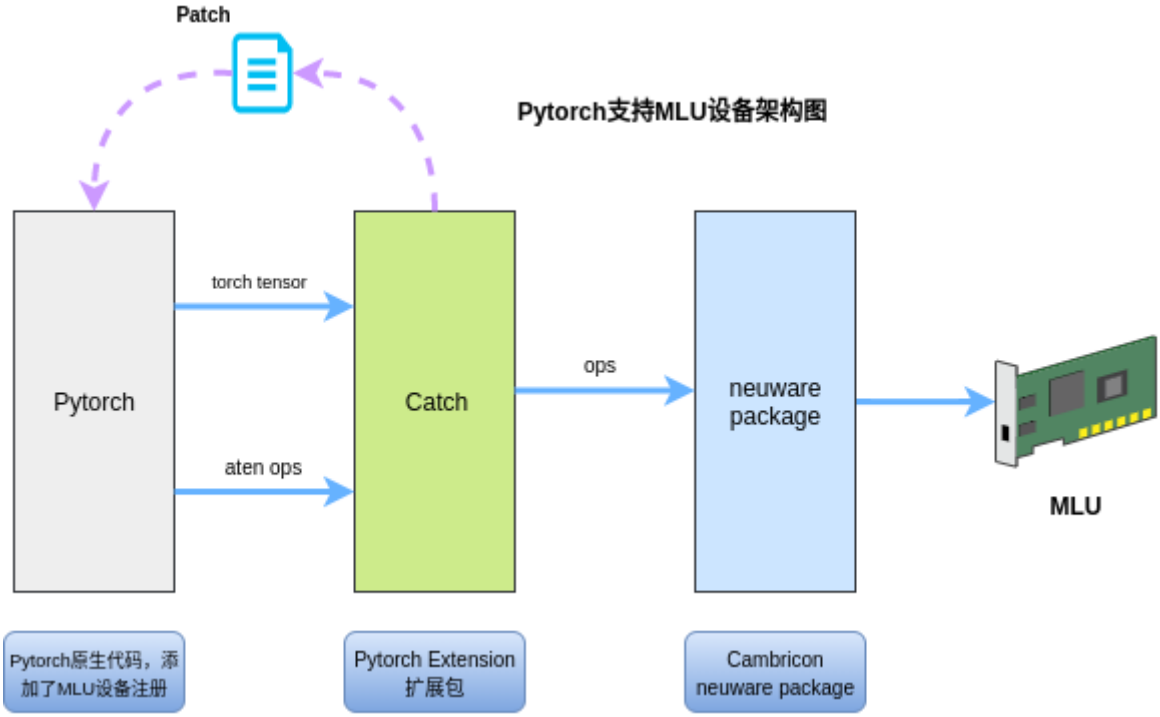
序号	主要特性支持
1	支持float, bfloat16数据类型
2	支持网络训练
3	支持网络推理
4	支持DDP分布式训练
5	支持图优化和图融合
6	支持AMP混合精度训练

- Google XLA所开源的代码、文档及其他如下：

序号	主要开源内容
1	源代码、脚本、测试用例及demo程序
2	docker
3	Cloud TPUs(云平台环境)
4	API_GUIDE.md
5	CODE_OF_CONDUCT.md
6	CONTRIBUTING.md
7	LICENSE
8	OP_LOWERING_GUIDE.md
9	README.md
10	TROUBLESHOOTING.md

三、系统架构

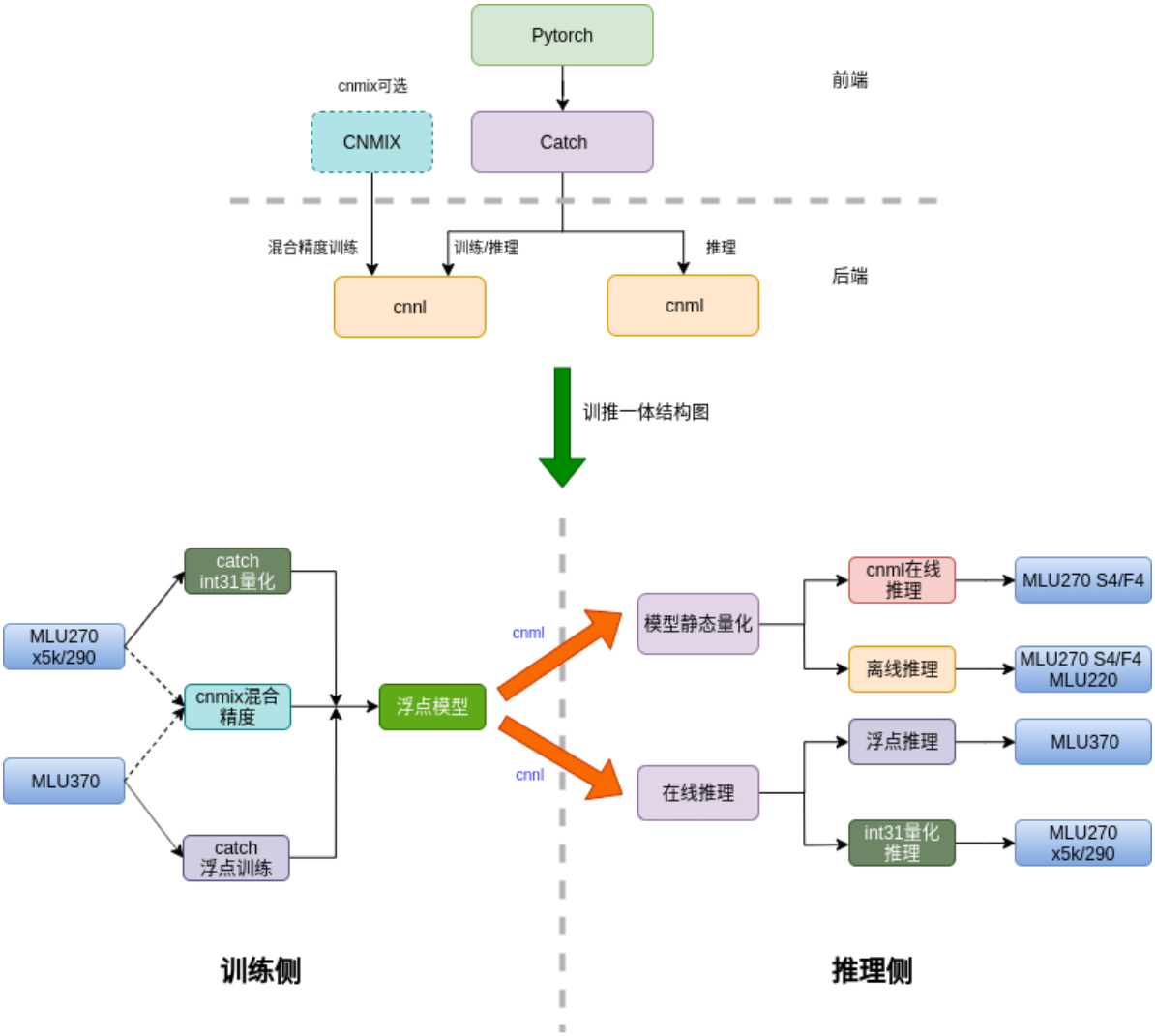
Catch开源方案中，为了保持组件功能的完整性，一是要支持训练和分布式训练功能，二是要支持训练后推理的功能，三是要做到与原生Pytorch代码充分解偶以方便后续的升级和维护，Catch整体软件开源架构如下图所示：



在开源架构图中，Catch以Pytorch Extension的形式来完成对MLU设备的支持，涉及到对原生Pytorch源码进行改动的地方，则以patch形式打到Pytorch仓库中。Catch通过借助neuware package所提供的高性能算子库及运行时库在MLU上加速神经网络的训练和推理过程。对于Catch自身的软件架构，考虑到当前Catch训练和推理软件栈支持的情况，以及后续Catch大概的产品形态，这里提出了四个开源方案供选择，具体架构图及优缺点如下所示：

1、方案一 ----- cnml+cnnl后端结构图

方案一中，Catch集成cnnl和cnml两个后端计算库，cnnl用于网络训练和推理，cnml用于网络推理。cnmix作为可选组件拓展Catch的功能，可以用来进行混合精度训练，加速训练的速度。



该方案优缺点具体如下。

优点：

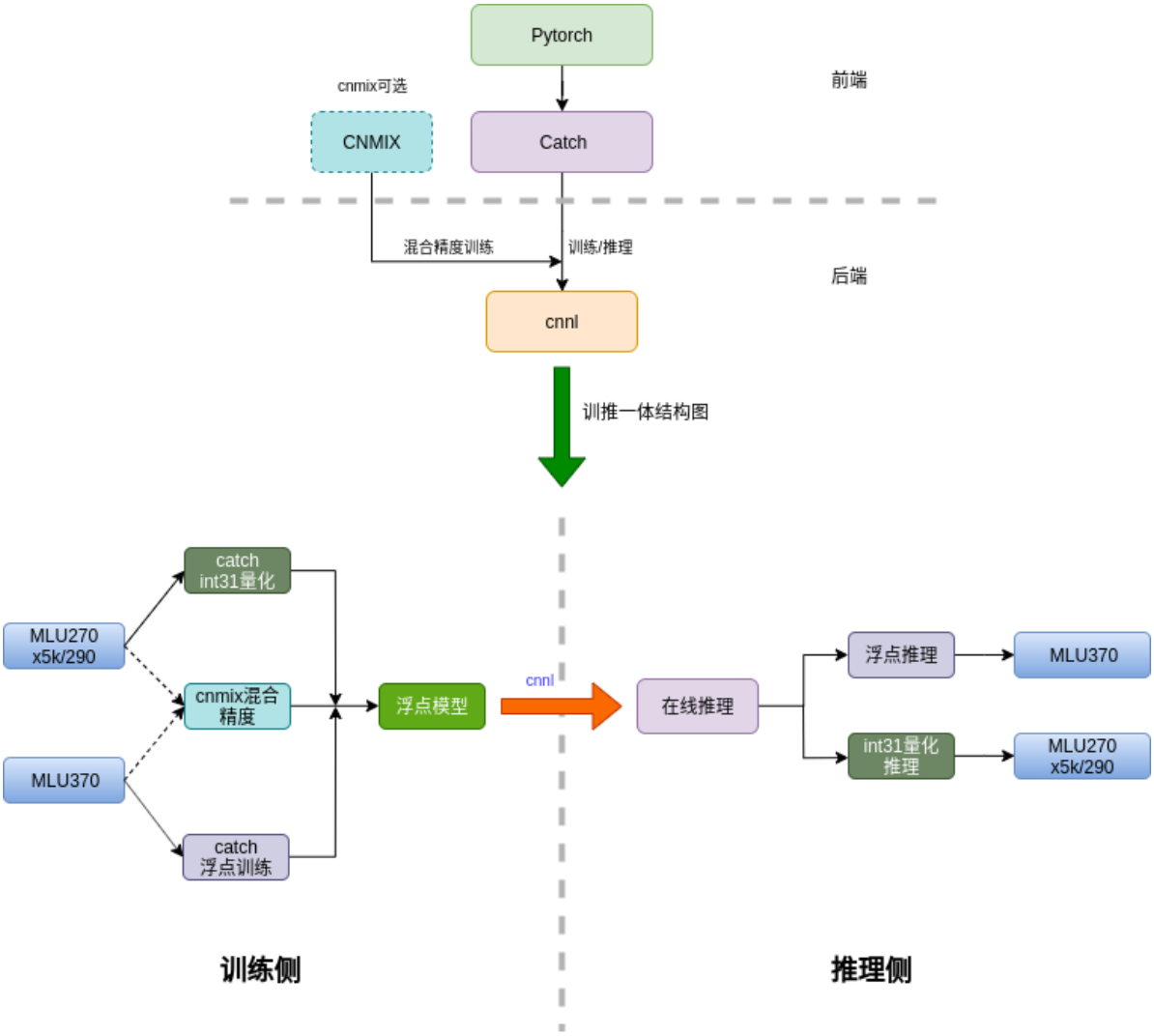
- cnnl在训练场景下相对比较成熟，cnml在推理场景下相比比较成熟，这会增加整个软件栈的稳定性;

缺点：

- 当前Catch训练是基于社区pytorch 1.6版本，而Catch推理是基于社区pytorch 1.3版本，pytorch 1.6上虽然有cnml推理相关的代码，但一直没有测试过。若采用此方案，需要pytorch 1.6正式支持cnml后端，这会带来不少研发和测试的工作量;
- 使用Catch固定int31做训练或者Catch + cnmix做混合精度训练后的模型，然后在cnml上做推理，这一块先前一直没有正式支持过，整条路也没有正式的趟通过;
- cnml组件后续会逐渐被magicmand所取代，在公司战略层面投入也逐渐减少，后续的维护投入可能存在不足;
- cnml暂不支持MLU270 X5K/MLU290/MLU370板卡，cnnl暂不支持MLU270 S4/F4卡，这导致训练和推理必须在不同的卡上进行，影响用户的使用;
- 在MLU270 X5K/MLU290上进行训练，为提高性能需要依赖cnmix，但cnmix中包含pytorch和tf两个框架的代码。这里存在两个问题，一是cnmix是否要开源还不确定，二是cnmix对tf有依赖，假如cnmix开源，但如何解决这个依赖问题还需要进一步考虑;

2、方案二 ----- cnnl后端结构图

方案二中，Catch只集成cnnl后端计算库，cnnl用于网络训练和推理。cnmix作为可选组件拓展Catch的功能，可以用来进行混合精度训练，加速训练的速度。



该方案优缺点具体如下。

优点：

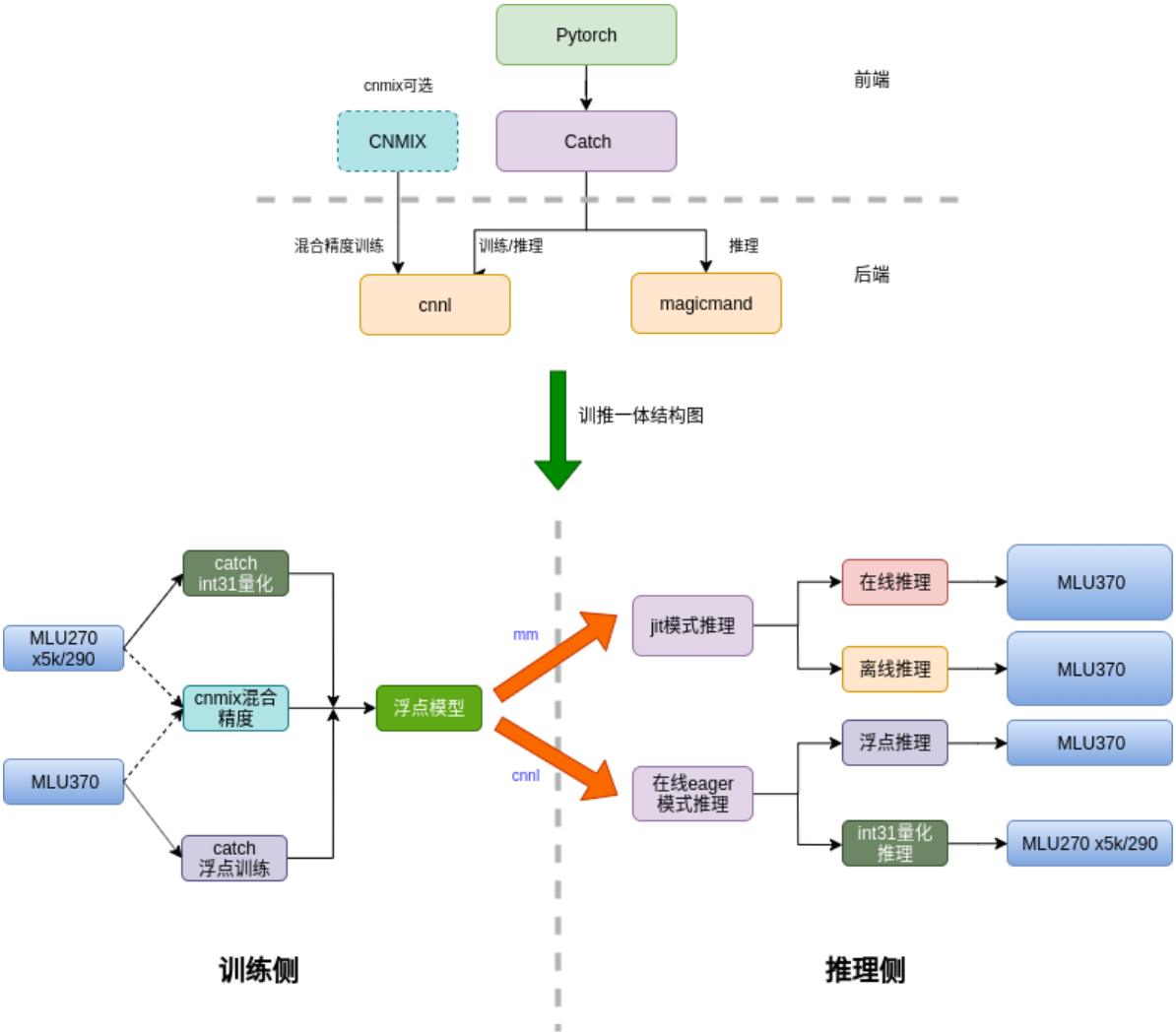
- 训练和推理都用同一个cnnl后端，可以保持软件栈以及板卡支持形态的统一性，避免训练和推理时用户在不同的板卡间进行切换;
- 对原生pytorch接口基本没有侵入，能够支持原生eager和jit模式训练和推理，这使得用户代码做很少的改动就能切到MLU上训练和推理;
- 可以保证训练和推理都是基于相同的社区pytorch 1.6版本;

缺点：

- 在MLU270 X5K/MLU290上使用固定int31位宽做训练和推理，网路性能比较差，这会影响用户的体验;
- cnml毕竟在MLU推理上做过不少的优化，先前公司所声称的推理性能都是基于cnml的，没有cnml这个组件，推理性能上可能对开源用户造成误解;
- 当期cnnl应该还不支持MLU270 S4/F4卡，这会导致所开源的Catch只能支持部分板卡;
- 在MLU270 X5K/MLU290上进行训练，为提高性能需要依赖cnmix，但cnmix中包含pytorch和tf两个框架的代码。这里存在两个问题，一是cnmix是否要开源还不确定，二是cnmix对tf有依赖，假如cnmix开源，但如何解决这个依赖问题还需要进一步考虑;

3、方案三 ----- cnnl+mm后端结构图

方案三中，Catch集成cnnl和mm两个后端计算库，cnnl用于网络训练和推理，mm用于网络推理。cnmix作为可选组件拓展Catch的功能，可以用来进行混合精度训练，加速训练的速度。



该方案优缺点具体如下。

优点：

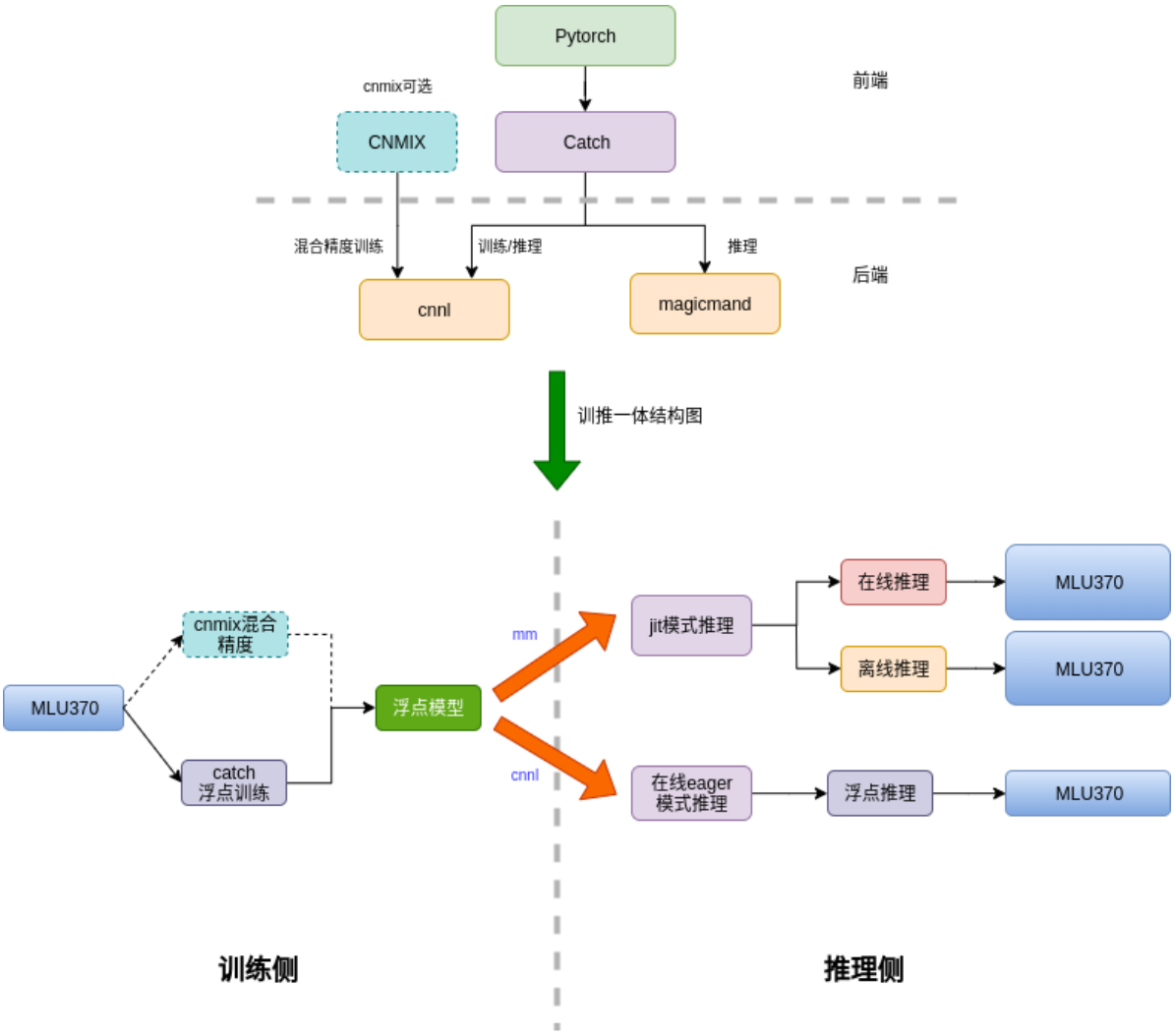
- Catch训练和mm推理都是基于cnnl后端，这对于开发和调试问题都比较方便;
- mm会逐渐替代cnml，在公司战略层面上会存在相当长的一段维护周期;
- 训练所支持的板卡，mm也都能支持，这可以保证训练和推理所使用板卡的统一性;
- 使用mm做推理，推理性能上应该能得到保证，并能跟公司宣称的性能上对齐;

缺点：

- mm目前尚不成熟，不经过几个版本的更新迭代而贸然开源，可能会引发比较多的问题;
- Catch暂没有适配mm后端，若适配的话会带来不少的开发和测试的工作量;
- 在MLU270 X5K/MLU290上进行训练，为提高性能需要依赖cnmix，但cnmix中包含pytorch和tf两个框架的代码。这里存在两个问题，一是cnmix是否要开源还不确定，二是cnmix对tf有依赖，假如cnmix开源，如何解决这个依赖问题还需要进一步考虑;

4、方案四 ----- cnnl+mm后端结构图(MLU370)

同方案三，该方案中Catch集成cnnl和mm两个后端计算库，cnnl用于网络训练和推理，mm用于网络推理。cnmix作为可选组件拓展Catch的功能，可以用来进行混合精度训练，加速训练的速度。与方案三区别之处在于开源时方案四先支持MLU370设备，其他设备后续再添加支持。



该方案优缺点具体如下。

优点：

- Catch训练和mm推理都是基于cnnl后端，这对于开发和调试问题都比较方便;
- mm会逐渐替代cnml，在公司战略层面上会存在相当长的一段维护周期;
- 训练所支持的板卡，mm也都能支持，这可以保证训练和推理所使用板卡的统一性;
- 使用mm做推理，推理性能上应该能得到保证，并能跟公司宣称的性能上对齐;
- 暂时只支持MLU370设备，可以先解除对cnmix组件的强依赖性，cnmix后续可根据需求单独选择是否开源;
- 在浮点训练时，Catch对原生Pytorch接口侵入很小，方便用户快速迁移到MLU设备;
- 当前mm只对MLU370/MLU220做了支持，可有效利用mm在MLU370上的推理性能;

缺点：

- mm目前尚不成熟，不经过几个版本的更新迭代而贸然开源，可能会引发比较多的问题;
- Catch暂没有适配mm后端，适配的话会带来不少的开发和测试的工作量;

对比以上四种方案，方案一没法做到训推一体及同一个板卡上支持训练和推理; 方案二受限于cnmix开源与否以及Catch在固定int31模式下训练和推理性能问题; 方案三一是受限于cnmix开源与否，二是受限于mm当前只对MLU370/MLU220做了支持，对其他板卡暂没做支持，三是受限于mm自身代码的成熟度; 方案四只受限于mm的成熟度，其他没有限制。综上来看，方案四是目前比较好的开源方式，能够满足特性拆解中的所有需求，下面的章节将根据方案四进行模块的设计和计划的排定。

四、接口设计

Catch开源过程中会尽可能的复用原生的接口，对于那些不得不新增或者改动的接口，会在后续开源方案具体实施的过程中再确定，这里后续再完善。

五、模块设计/模块交互

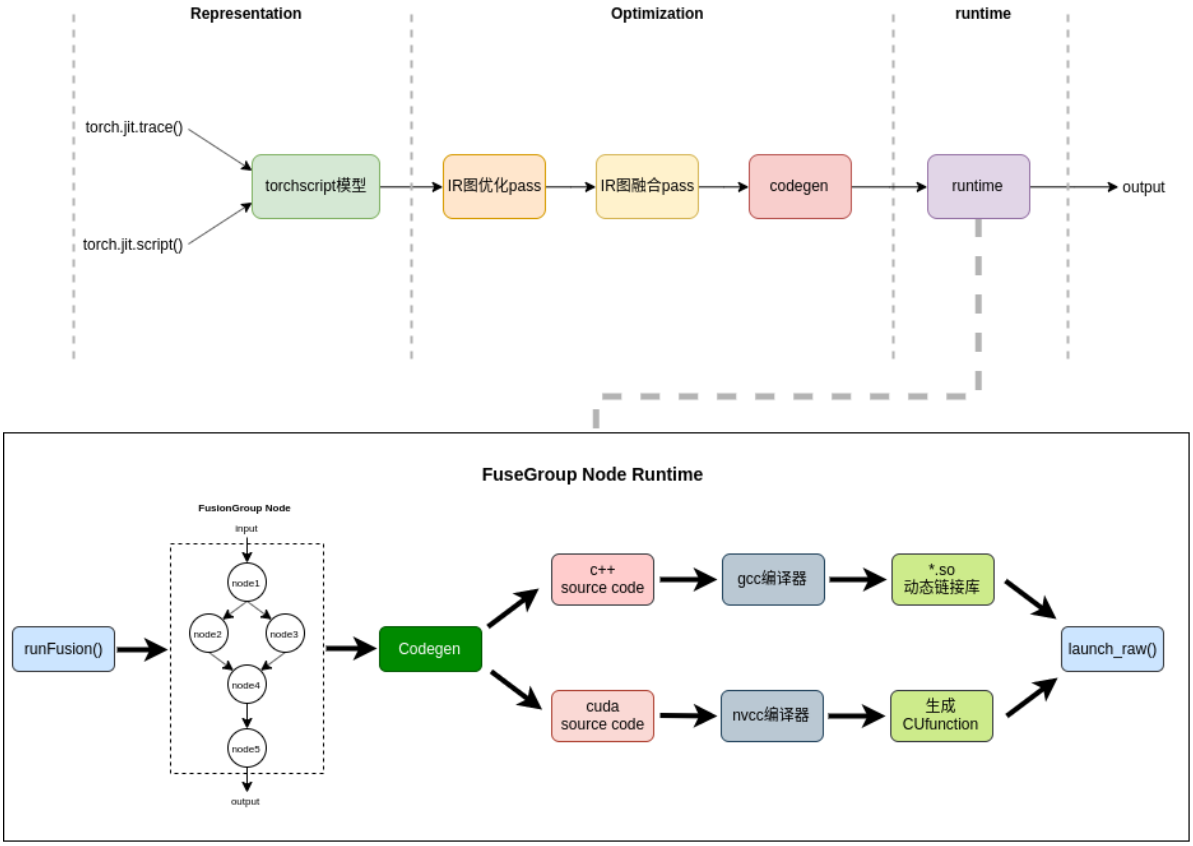
对于方案四中所展现的系统架构，训练相关的功能在Catch中已经支持，但推理相关的功能Catch还没有添加支持，因此开源方案中很大一部分工作在于对mm推理功能的支持。

1、Catch适配mm模块结构图

对于mm推理功能的支持，这里有以下几种方案可供选择：

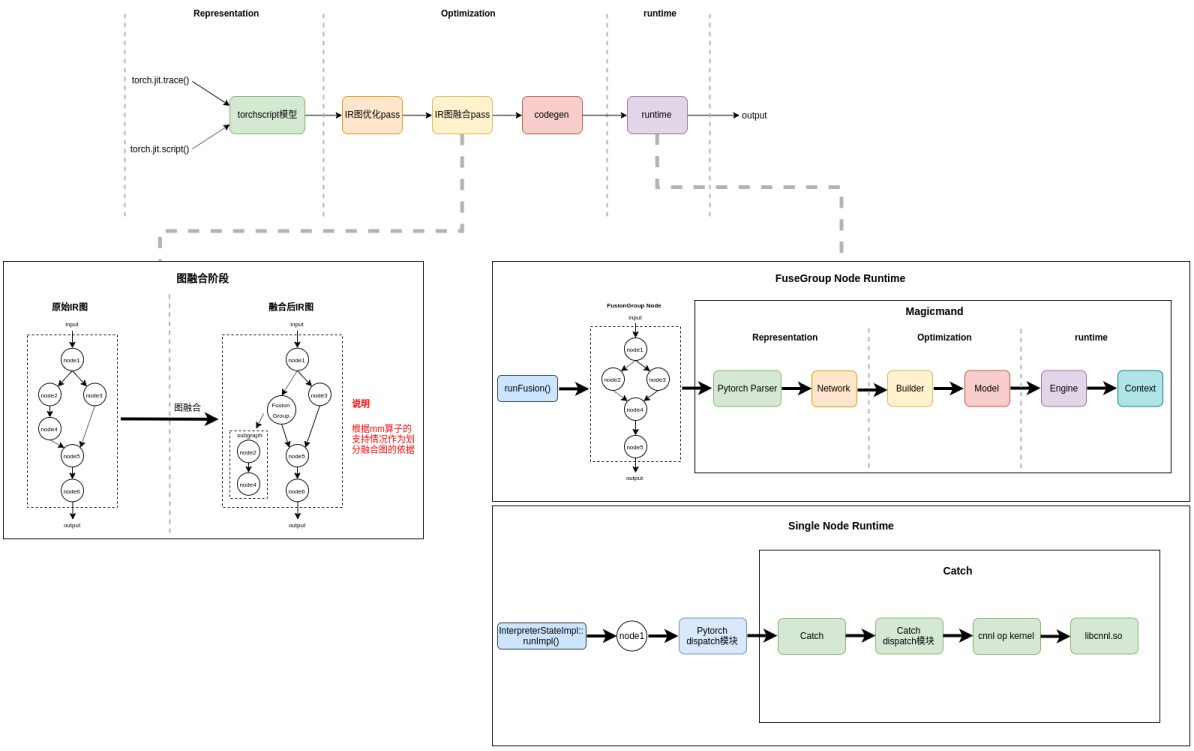
1、方案一 ----- Catch适配mm parser

Catch适配mm parser的方式类似于原生Pytorch jit融合模式做推理的形式，原生Pytorch jit融合模式运行机制如下图所示：



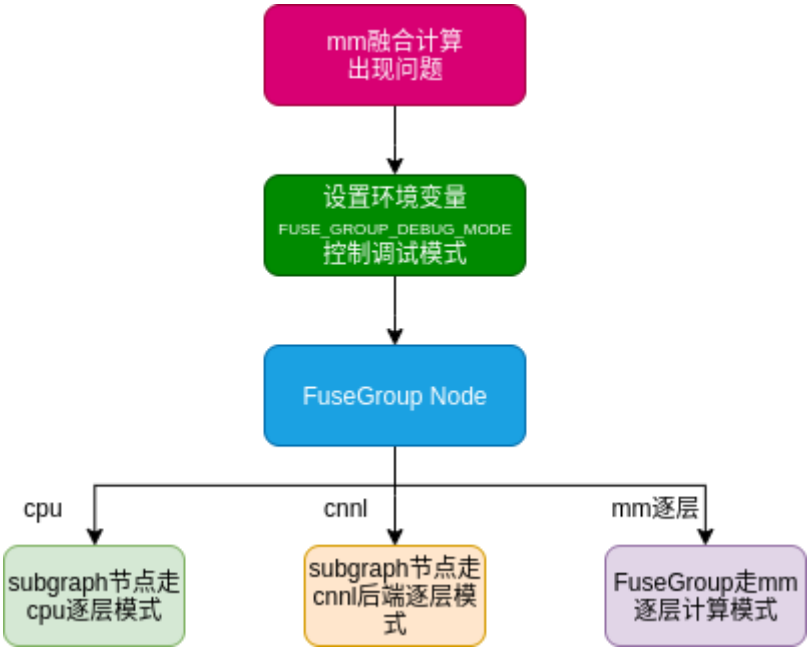
原生Pytorch jit运行机制中，图融合阶段会将可以融合的节点放在一个FusionGroup节点中，在runtime阶段执行FusionGroup节点的操作函数时，Codegen模块会编译FusionGroup中所包含的subgraph并生成对应设备的source code，然后再由gcc或nvcc编译器将source code编译成可执行的指令码。对于不能融入FusionGroup中的节点，runtime会直接调用aten中的算子后端完成计算过程。

以mm parser的形式进行适配理论上是最快的适配方式，pytorch不需要像以前cnml的方式一个一个算子进行适配，只需要在图融合阶段将整个IR图中mm可支持的节点放在若干个FusionGroup中，每个FusionGroup在执行的时候会调用mm提供的相关接口完成子图的优化、编译和运行等阶段，具体流程如下图所示：



Catch集成mm parser的方式跟原生Pytorch jit图融合机制很相似，整个mm就类似于原生Pytorch中Codegen+gcc或Codegen+nvcc模块，完成图优化、图编译和运行等功能。另外，对于eager运行模式，只需要支持唯一的cnnl后端，不再添加mm算子api的适配。

当FuseGroup融合计算出现问题时，对于FuseGroup融合模式的调试，可通过设置FUSE_GROUP_DEBUG_MODE环境变量控制FuseGroup中计算图的执行模式来达到调试的目的。该环境变量可设置三种调试模式：cpu模式、cnnl后端模式以及mm逐层计算模式。



另外，借助于catch中已支持的fallback-to-cpu的功能，该方案还有一个很大的优点是可以实现fallback-to-cpu的功能，当算子既不能被mm支持，也不能被cnnl支持时，可以自动将算子调用fallback到cpu上执行，这有效的保证了用户网络的正常执行，方便用户快速的对网络进行验证。

该方案优缺点具体如下。

优点：

- 比较简单，开发工作量少，只需要将IR图传给mm parser即可;

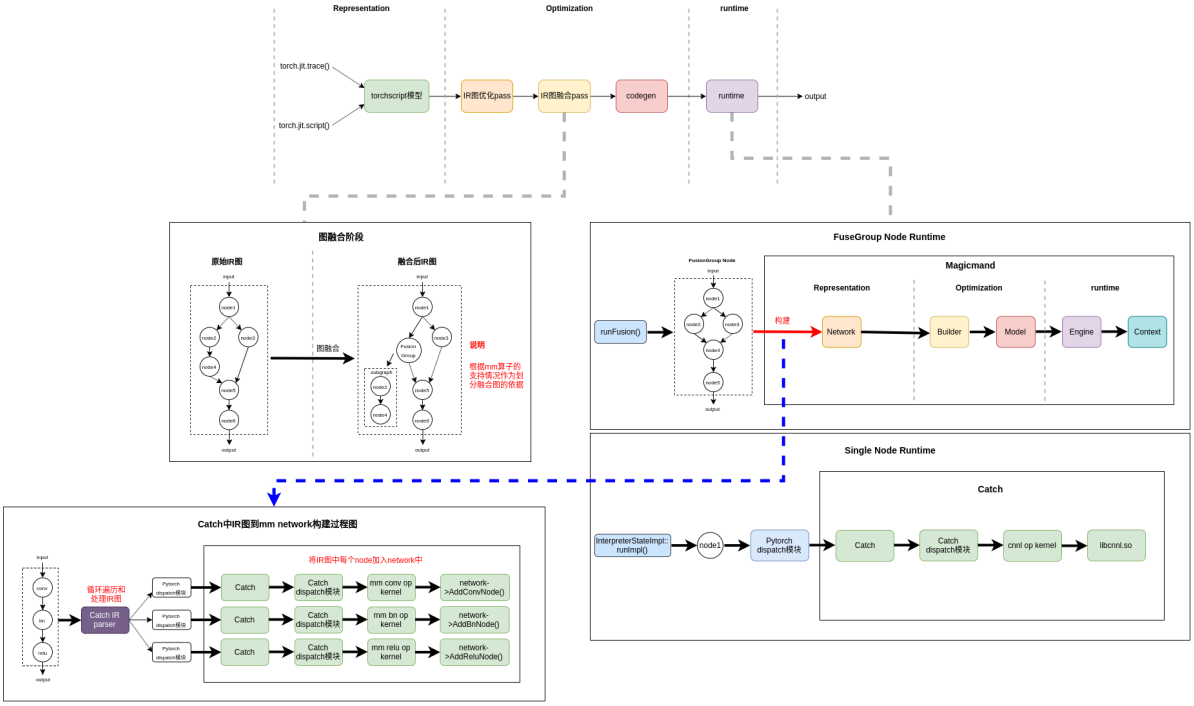
- 跟原生pytorch jit融合的工作机制很相似;
- 对原生pytorch无接口侵入，方便用户切换到mlu上做推理;
- 支持自动fallback功能，能够保证网络的正常执行;

缺点：

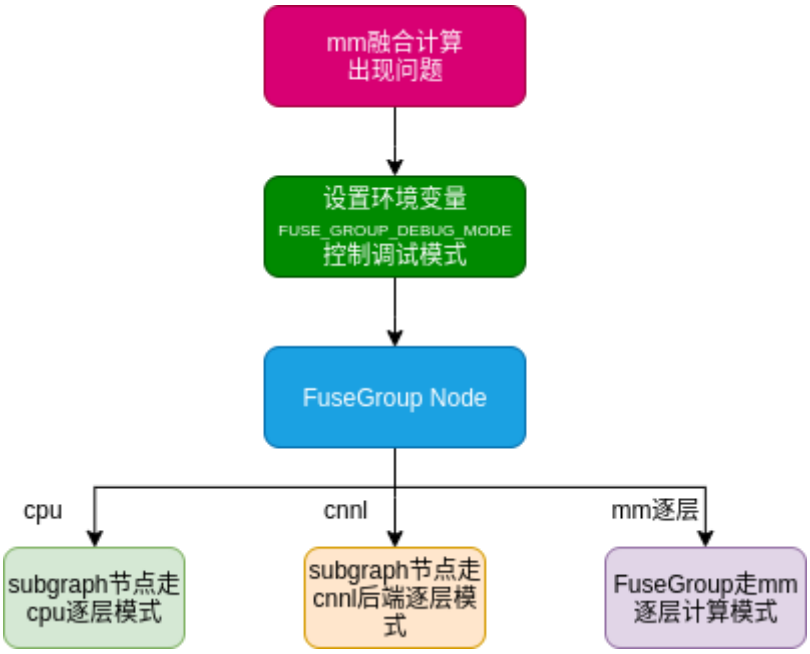
- mm paser使用的是pytorch原生的libtorch.so，而Catch使用的是cambricon对pytorch修改后编译生成的libtorch.so，若cambricon生成的libtorch.so对原生的接口及功能有改动，会对mm parser产生影响，所以这里必须要求cambricon对pytorch的改动不能破坏原生的接口和功能;

2、方案二 ----- Catch适配mm引擎API(eager走cnnl后端)

Catch适配mm引擎api的方式类似于Catch适配cnml算子的方式，mm提供了算子接口供用户使用。方案二相比方案一来说，相当于把mm parser所做的事情挪到了Catch中，由Catch来完成Pytorch IR到mm network的转换，这样就可以解决方案一中libtorch.so库潜在冲突的问题。



当FuseGroup融合计算出现问题时，对于FuseGroup融合模式的调试，可通过设置FUSE_GROUP_DEBUG_MODE环境变量控制FuseGroup中计算图的执行模式来达到调试的目的。该环境变量可设置三种调试模式：cpu模式、cnnl后端模式以及mm逐层计算模式。



另外，跟方案一样，该方案也能支持自动fallback-to-cpu功能。

该方案优缺点具体如下。

优点：

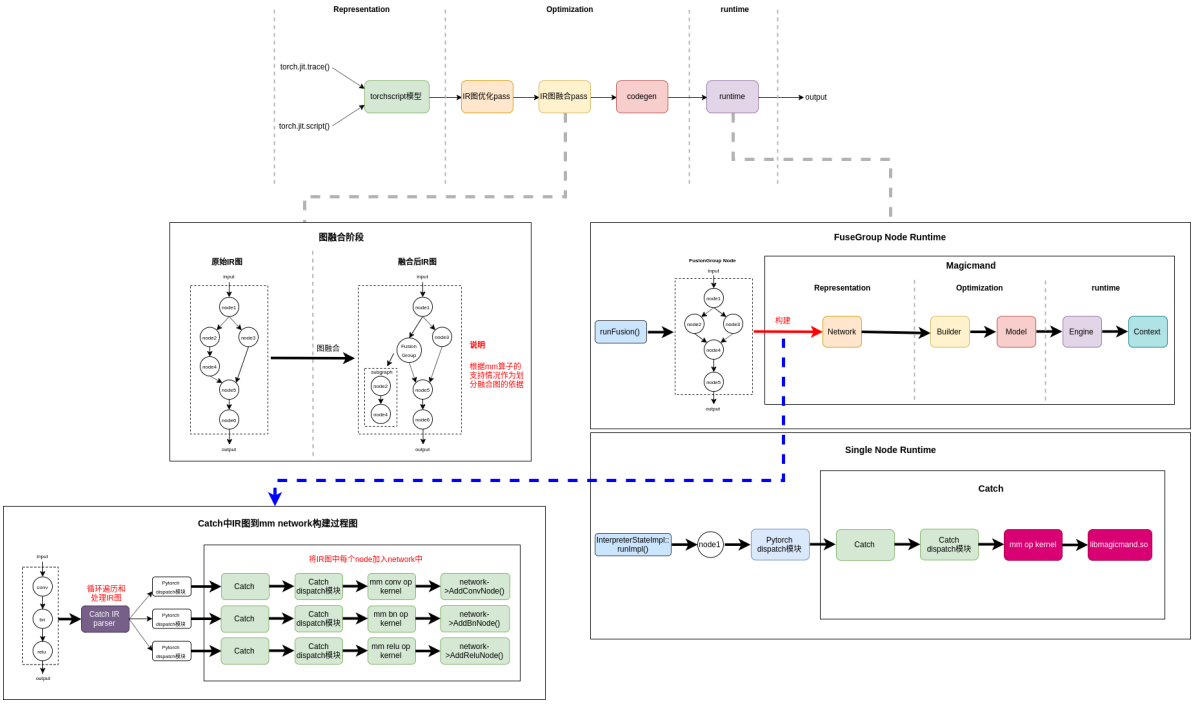
- 可以解决方案一中mm parser与Catch间libtorch.so存在潜在冲突的问题;
- 跟原生pytorch jit融合的工作机制很相似，计算后端比较单一;
- 对原生pytorch无接口侵入，方便用户切换到mlu上做推理;
- 支持自动fallback功能，能够保证网络的正常执行;

缺点：

- 工作量相比方案一有不少的增加，基本上是在pytorch框架中重新做了一遍mm pytorch parser的工作;

3、方案三 ----- Catch适配mm引擎API(eager走mm后端)

方案三相比方案二来说，eager模式除已经支持的cnnl后端外，还增加了mm后端算子的支持，这样可以用eager模式对mm进行调试，具体结构图如下所示：



方案三中具体可总结成以下几点：

- 在训练模式下，eager和jit都走cnnl后端;
- 在推理模式下，eager和jit trace默认走cnnl后端;
- torchscript运行时能融合的子图自动走mm，不能融合的节点仍走cnnl后端;
- 若需要调试mm融合问题，可通过开关控制eager模式走mm后端;

该方案优缺点具体如下。

优点：

- 可以解决mm parser与Catch间libtorch.so潜在冲突的问题;
- 跟原生pytorch jit融合的工作机制很相似;
- 当mm融合出现问题时，可以开启mm后端，用eager模式进行调试;

缺点：

- 工作量相比方案一有不少的增加;

- 需要定义接口来控制走cnnl后端还是走mm引擎api后端，对原生pytorch的使用有接口侵入;
- eager模式增加了一种MLU后端算子库，导致整个软件栈略显混乱一些，容易导致用户误使用cnnl或mm后端;

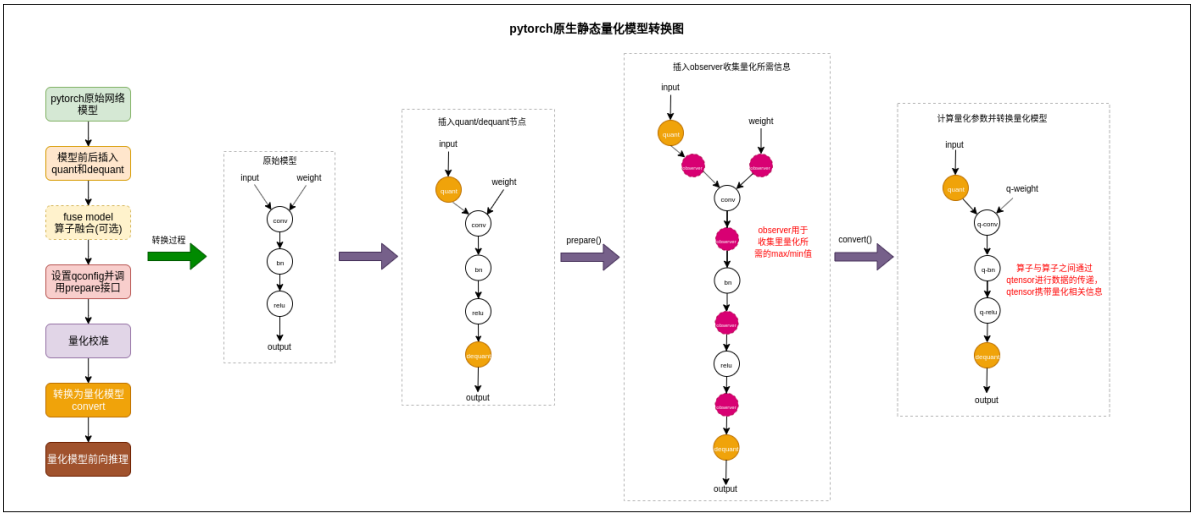
从以上三种方案来看，方案二虽然具有方案一的优点且能解决libtorch潜在冲突的问题，但工作量与mm pytorch parser重复太大，会造成很大的人力浪费。方案三虽然能解决libtorch.so潜在冲突以及解决mm融合调试的问题，但是会增加多个MLU后端算子计算库，容易导致用户误使用cnnl或mm后端，整个软件栈略显混乱。方案一虽然有libtorch.so冲突的隐患，但只要单元测试充分，是可以避免这个问题的，并且方案一更贴合原生pytorch的机制，整个软件栈也比较清晰，因此，采用方案一来进行mm的适配会更好一些。

2、Catch推理量化模式支持

为了发挥MLU板卡更好的推理性能，Catch除支持最基础的浮点推理之外，还需要支持定点推理。而对于定点推理，原生Pytorch提供了三种定点计算支持模式：训练后静态量化、训练后动态量化以及QAT量化感知训练。对于定点模式的支持，最好的方式是无缝的复用原生Pytorch已提供的方法，因为这样会更符合社区用户的使用习惯。在介绍Catch开源所采用的定点支持方案之前，有必要先对原生pytorch的定点支持方式进行介绍，具体如下所示：

- 训练后静态量化

对于训练后静态量化模式，在前向推理之前，需要先得到activation和weight对应的量化信息，然后在推理时就可以用量化信息对输入和weight进行量化，静态量化所支持的量化位宽为：qint8, quint8, qint32。原生Pytorch支持静态量化的算子可参考：pytorch/torch/quantization/default_mappings.py中的DEFAULT_MODULE_MAPPING表，具体量化流程如下图所示：

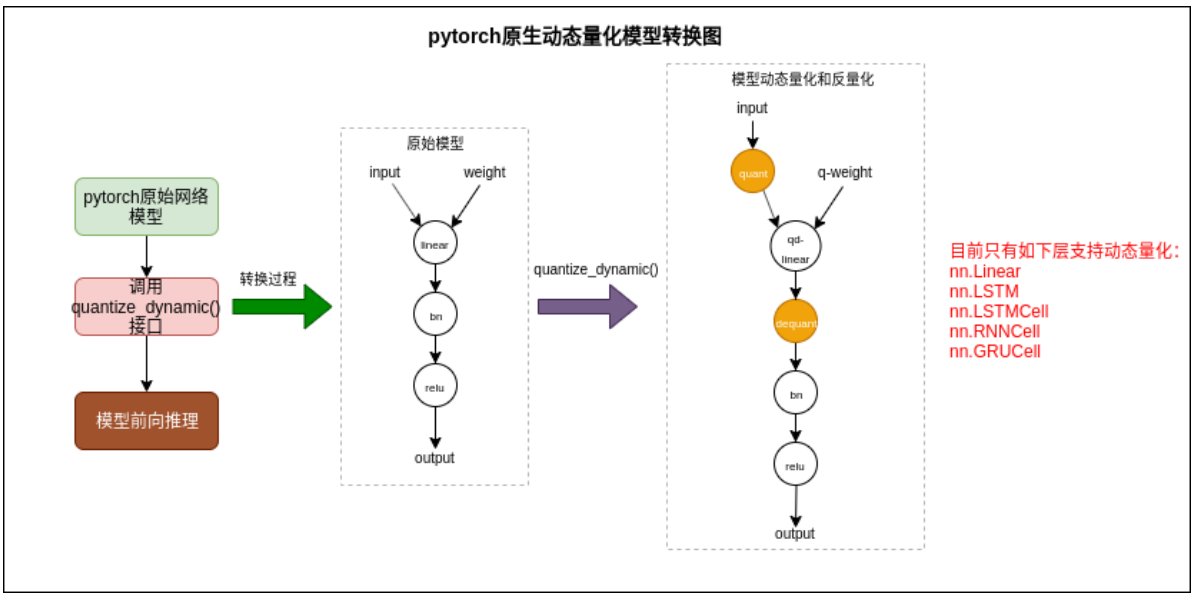


- 训练后动态量化

Pytorch原生动态量化流程如下图所示，相比静态量化主要区别点在于：

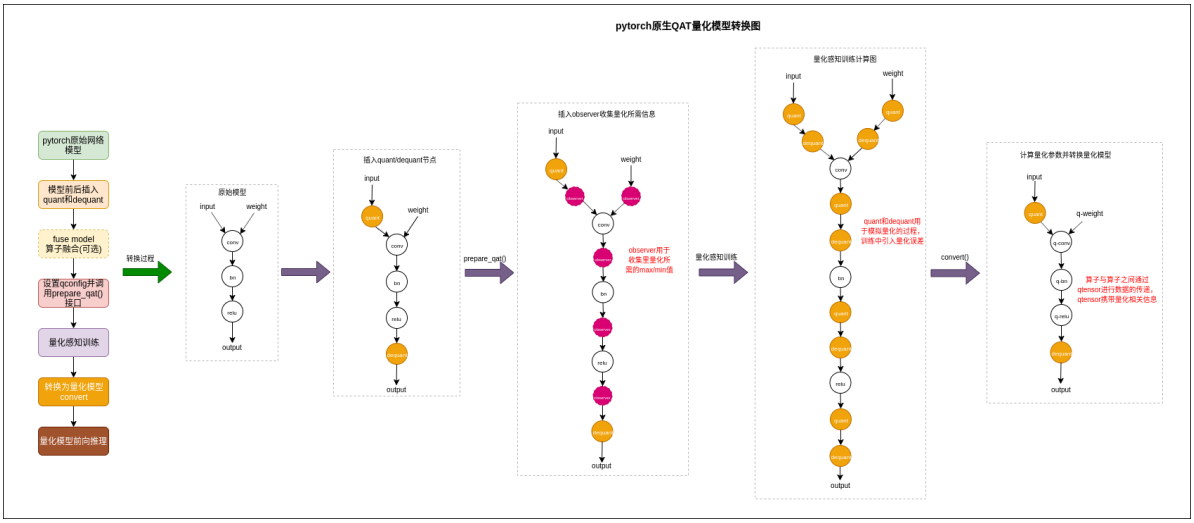
1. 静态量化的float输入必须经过Quant操作变为int，此后整个网络到输出之前都是定点类型传输;
2. 动态量化的float输入是经过动态计算的scale和offset量化为定点类型，op在输出时会转回float类型。

原生Pytorch目前支持动态量化的层可参考：pytorch/torch/quantization/default_mappings.py中的DEFAULT_DYNAMIC_MODULE_MAPPING



- QAT量化感知训练

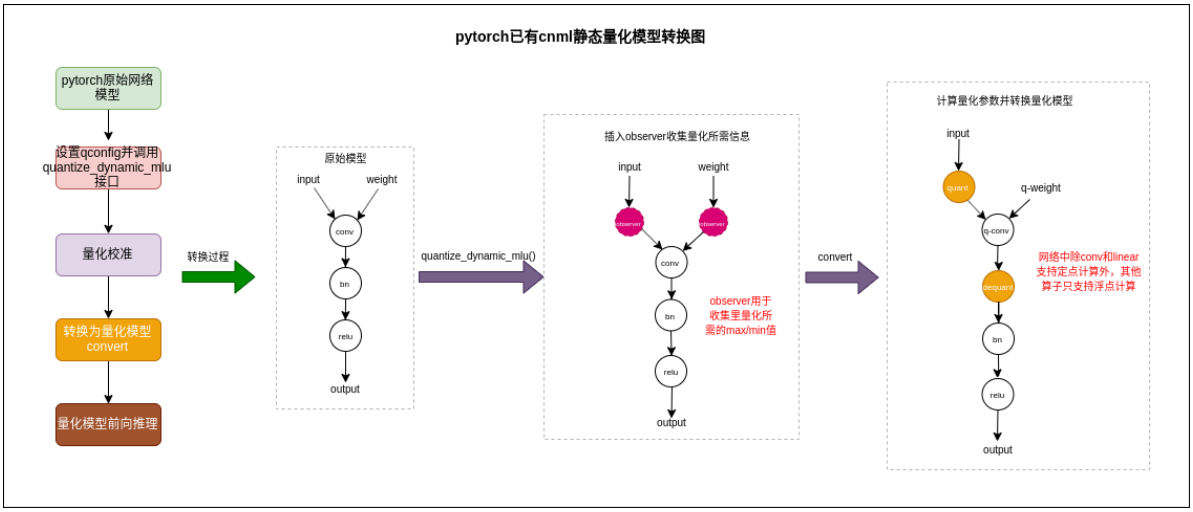
QAT会在训练过程中开启伪量化操作，将量化误差带入到训练过程中。在QAT过程中会获取到待量化层的量化信息，并根据这些信息计算出scale和offset参数，QAT做完之后通过调用convert()函数将QAT模型转成静态量化模型，这样就可以用已有的量化参数做静态量化推理了，具体流程如下图所示：



原生Pytorch目前支持QAT量化的层可参考：pytorch/torch/quantization/default_mappings.py中的DEFAULT_QCONFIG_PROPAGATE_WHITE_LIST

- cnml时代静态量化支持方式

对于MLU设备来说，目前只有Conv和Linear算子支持量化，其他算子支持浮点类型计算。cnml静态量化方式整体更类似于原生Pytorch的动态量化方式，只有conv和linear支持量化，其他算子运行浮点模式，具体流程如下图所示：



对比原生Pytorch的三种量化方式可知，对于MLU设备来说，原生量化功能支持与不支持情况如下所示：

1. MLU设备可支持原生动态量化训练，QAT量化感知训练中的训练部分;
2. MLU设备不能支持原生静态量化方式，因为MLU设备目前只支持conv/linear等算子的量化;
3. 目前MLU设备支持int8, int16, int31量化方式，而原生Pytorch支持int8, uint8, int32量化方式，这意味着MLU设备只能支持原生int8量化方式，对其他量化方式暂不支持;
4. mm目前还不支持原生pytorch量化所对应的IR图;

从上面的分析来看，对原生Pytorch静态量化的支持目前各组件都不太成熟，而cnml所采用的静态量化方式虽然有接口上的侵入，但cambricon各软件栈对这种方式支持情况更加的成熟，所以开源方案中分两部来做：

1. 先支持先前cnml所采用的静态量化方式，做到向前兼容，保证用户先前的脚本可以正常跑;
2. 待cambricon各软件栈对原生pytorch静态量化支持的更加成熟后，在添加对原生量化方式的支持;

3、Catch训练相关模块支持

为更好的展现Catch训练功能，需要对Catch中的一些关键模块的代码和组件进行开源，具体需要开源的模块和组件如下表格所示：

关键模块/组件	是否开源
Catch仓库	是
torchvision仓库	是
pytorch_models仓库	是(先开源部分网络)
cnnl后端kernel	是
ddp模块	是
cache allocator模块	是
debug allocator模块	是
queue管理模块	是
异步launch模块	是
device管理模块	是
调试工具	是
pin memory模块	是
算子分发模块	是
bangc混合编译模块	是
摆数统一相关模块	是
单元测试相关	是
benchmark相关	是

六、开源计划和人力预估

Catch开源具体需要做的事情如下表所示：

开源前需要做的事情	详细描述	人力预计投入
开源方案设计	完成开源方案整体设计	4人周
专利/软著编写	专利和软著的审核流程比较长，需要先开展专利和软著的挖掘和编写	4人周
依赖库开源协议检查	对Catch/pytorch_models所依赖库的开源协议进行检查，确认是否能够商用使用协议	0.3人周
训练相关开源代码审核	确定训练需要开源的算子、功能模块、demo程序、网络、工具、脚本等，并协助法务做侵权分析	3人周
Catch推理适配mm	Catch通过适配mm增加推理功能支持	4人周
Catch推理算子单元测试	使用mm后端，完成pytorch推理相关算子测例的编写	20人周(暂定50个算子支持)
Catch推理量化模式支持	Catch推理需要支持量化功能，以提升推理网络的性能	3人周
Catch推理demo程序支持	确定推理所需要支持的网络并提供相应的demo	4人周(暂定10个网络支持)
推理相关开源代码审核	确定推理需要开源的算子、功能模块、demo程序、网络、工具、脚本等，并协助法务做侵权分析	2人周
训练相关代码整理	1.代码风格统一 2.日志信息检查，编译warning 3.关键字检查，如：厂商等 4.冗余代码清理 5.添加文件法律信息	4人周
推理相关代码整理	1.代码风格统一 2.日志信息检查，编译warning 3.关键字检查，如：厂商等 4.冗余代码清理 5.添加文件法律信息	2人周
底层库开源相关	跟底层协商，确认所依赖的底层库以及开源的版本	1人周
代码审核	组织代码评审会议，对算子，demo，工具，网络等代码进行审核	2人周
开源代码内部回归测试	1.确认pytorch框架cpu/gpu需要测试的功能 2.确认Catch需要测试的功能 3.单元测试的完善和测试	2人周
开源文档编写和审核	编写开源文档，并提交文档组进行审核	2人周
外网服务器测试环境搭建	搭建外网服务器环境	1人周

开源前需要做的事情	详细描述	人力预计投入
外网环境下测试	在外网环境下完成开源代码所有的回归测试	1人周
外网Precheckin/daily测试环境搭建	搭建外网precheckin和daily测试环境，方便开发者进行代码开发和上传	1人周
代码上传和维护	代码上传到github并维护	
总计		59.3人周

七、备选方案

暂无备选方案

八、测试方案

具体测试内容在开源方案实施阶段再进一步细化，这里先指出Catch开源需要覆盖如下内容：

- 测试内容需要覆盖所有算子、功能模块、demo以及网络;
- 测试能够做到自动化，方便外部开发者提交代码;

九、遗留问题

无