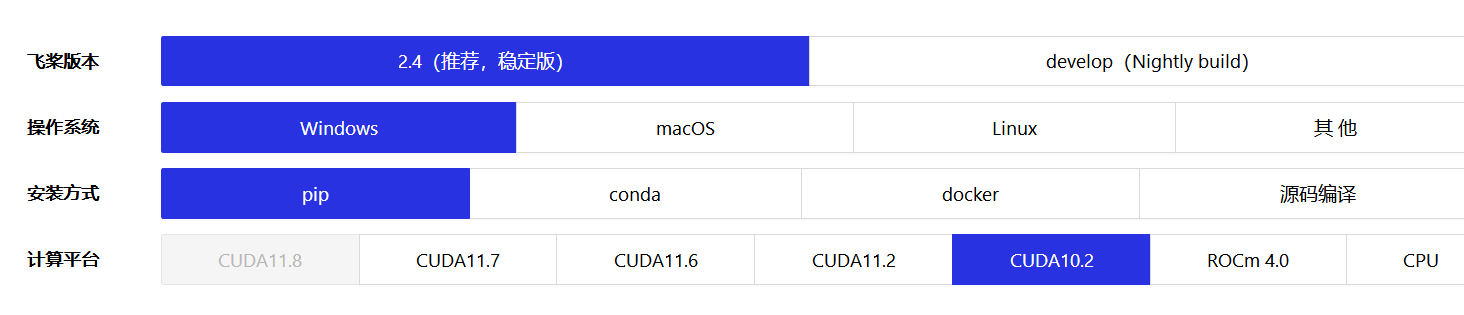
安装PaddlePaddle包



简单地说，深度学习任务一般分为以下几个核心步骤：

数据集定义与加载

飞桨在 paddle.vision.datasets 下内置了计算机视觉（Computer Vision，CV）领域常见的数据集，如 MNIST、Cifar10、Cifar100、FashionMNIST 和 VOC2012 等。在本任务中，先后加载了 MNIST 训练集（mode='train'）和测试集（mode='test'），训练集用于训练模型，测试集用于评估模型效果。

模型组网

飞桨的模型组网有多种方式，既可以直接使用飞桨内置的模型，也可以自定义组网

模型训练与评估

模型训练

模型训练需完成如下步骤：

使用 paddle.Model 封装模型。 将网络结构组合成可快速使用 飞桨高层 API 进行训练、评估、推理的实例，方便后续操作。

使用 paddle.Model.prepare 完成训练的配置准备工作。 包括损失函数、优化器和评价指标等。飞桨在 paddle.optimizer 下提供了优化器算法相关 API，在 paddle.nn Loss层 提供了损失函数相关 API，在 paddle.metric 下提供了评价指标相关 API。

使用 paddle.Model.fit 配置循环参数并启动训练。 配置参数包括指定训练的数据源 train\_dataset、训练的批大小 batch\_size、训练轮数 epochs 等，执行后将自动完成模型的训练循环。

因为是分类任务，这里损失函数使用常见的 CrossEntropyLoss （交叉熵损失函数），优化器使用 Adam，评价指标使用 Accuracy 来计算模型在训练集上的精度

模型评估

模型训练完成之后，调用 paddle.Model.evaluate ，使用预先定义的测试数据集，来评估训练好的模型效果，评估完成后将输出模型在测试集上的损失函数值 loss 和精度 acc

模型推理

模型保存

模型训练完成后，通常需要将训练好的模型参数和优化器等信息，持久化保存到参数文件中，便于后续执行推理验证。

在飞桨中可通过调用 paddle.Model.save 保存模型

模型加载并执行推理

执行模型推理时，可调用 paddle.Model.load 加载模型，然后即可通过 paddle.Model.predict\_batch 执行推理操作

Tensor 的概念介绍

飞桨使用张量（Tensor） 来表示神经网络中传递的数据，Tensor 可以理解为多维数组，类似于 Numpy 数组（ndarray） 的概念。与 Numpy 数组相比，Tensor 除了支持运行在 CPU 上，还支持运行在 GPU 及各种 AI 芯片上，以实现计算加速；此外，飞桨基于 Tensor，实现了深度学习所必须的反向传播功能和多种多样的组网算子，从而可更快捷地实现深度学习组网与训练等功能。两者具体异同点可参见下文 Tensor 与 Numpy 数组相互转换。

深度学习模型需要大量的数据来完成训练和评估，这些数据样本可能是图片（image）、文本（text）、语音（audio）等多种类型，而模型训练过程实际是数学计算过程，因此数据样本在送入模型前需要经过一系列处理，如转换数据格式、划分数据集、变换数据形状（shape）、制作数据迭代读取器以备分批训练等。

在飞桨框架中，可通过如下两个核心步骤完成数据集的定义与加载：

定义数据集：将磁盘中保存的原始图片、文字等样本和对应的标签映射到 Dataset，方便后续通过索引（index）读取数据，在 Dataset 中还可以进行一些数据变换、数据增广等预处理操作。在飞桨框架中推荐使用 paddle.io.Dataset 自定义数据集，另外在 paddle.vision.datasets 和 paddle.text 目录下飞桨内置了一些经典数据集方便直接调用。

迭代读取数据集：自动将数据集的样本进行分批（batch）、乱序（shuffle）等操作，方便训练时迭代读取，同时还支持多进程异步读取功能可加快数据读取速度。在飞桨框架中可使用 paddle.io.DataLoader 迭代读取数据集。

使用 paddle.io.Dataset 自定义数据集

在实际的场景中，一般需要使用自有的数据来定义数据集，这时可以通过 paddle.io.Dataset 基类来实现自定义数据集。

可构建一个子类继承自 paddle.io.Dataset ，并且实现下面的三个函数：

\_\_init\_\_：完成数据集初始化操作，将磁盘中的样本文件路径和对应标签映射到一个列表中。

\_\_getitem\_\_：定义指定索引（index）时如何获取样本数据，最终返回对应 index 的单条数据（样本数据、对应的标签）。

\_\_len\_\_：返回数据集的样本总数。

使用 paddle.io.DataLoader 定义数据读取器

通过前面介绍的直接迭代读取 Dataset 的方式虽然可实现对数据集的访问，但是这种访问方式只能单线程进行并且还需要手动分批次（batch）。在飞桨框架中，推荐使用 paddle.io.DataLoader API 对数据集进行多进程的读取，并且可自动完成划分 batch 的工作。

飞桨框架目前在 paddle.vision.models 下内置了计算机视觉领域的一些经典模型，只需一行代码即可完成网络构建和初始化，适合完成一些简单的深度学习任务，满足深度学习初阶用户感受模型的输入和输出形式、了解模型的性能。

经典模型可以满足一些简单深度学习任务的需求，然后更多情况下，需要使用深度学习框架构建一个自己的神经网络，这时可以使用飞桨框架 paddle.nn 下的 API 构建网络，该目录下定义了丰富的神经网络层和相关函数 API，如卷积网络相关的 Conv1D、Conv2D、Conv3D，循环神经网络相关的 RNN、LSTM、GRU 等，方便组网调用

构建顺序的线性网络结构时，可以选择该方式，只需要按模型的结构顺序，一层一层加到 paddle.nn.Sequential 子类中即可。

构建一些比较复杂的网络结构时，可以选择该方式，组网包括三个步骤：

创建一个继承自 paddle.nn.Layer 的类；

在类的构造函数 \_\_init\_\_ 中定义组网用到的神经网络层（layer）；

在类的前向计算函数 forward 中使用定义好的 layer 执行前向计算

在准备好数据集和模型后，就可以将数据送入模型中启动训练评估了，概括地讲包括如下几步：

模型训练：训练包括多轮迭代（epoch），每轮迭代遍历一次训练数据集，并且每次从中获取一小批（mini-batch）样本，送入模型执行前向计算得到预测值，并计算预测值（predict\_label）与真实值（true\_label）之间的损失函数值（loss）。执行梯度反向传播，并根据设置的优化算法（optimizer）更新模型的参数。观察每轮迭代的 loss 值减小趋势，可判断模型训练效果。

模型评估：将测试数据集送入训练好的模型进行评估，得到预测值，计算预测值与真实值之间的损失函数值（loss），并计算评价指标值（metric），便于评估模型效果。

模型推理：将待验证的数据（样本）送入训练好的模型执行推理，观察并验证推理结果（标签）是否符合预期。

飞桨框架提供了两种训练、评估与推理的方法：

使用飞桨高层 API：先用 paddle.Model 对模型进行封装，然后通过 Model.fit 、 Model.evaluate 、 Model.predict 等完成模型的训练、评估与推理。该方式代码量少，适合快速上手。

使用飞桨基础 API：提供了损失函数、优化器、评价指标、更新参数、反向传播等基础组件的实现，可以更灵活地应用到模型训练、评估与推理任务中，当然也可以很方便地自定义一些组件用于相关任务中。

模型训练时，需要用到 CPU、 GPU 等计算处理器资源，由于飞桨框架的安装包是区分处理器类型的，默认情况下飞桨框架会根据所安装的版本自动选择对应硬件，比如安装的 GPU 版本的飞桨，则自动使用 GPU 训练模型，无需手动指定。因此一般情况下，无需执行此步骤。

但是如果安装的 GPU 版本的飞桨框架，想切换到 CPU 上训练，则可通过 paddle.device.set\_device 修改。如果本机有多个 GPU 卡，也可以通过该 API 选择指定的卡进行训练，不指定的情况下则默认使用 'gpu:0'。

用 paddle.Model 完成模型的封装后，需通过 Model.prepare 进行训练前的配置准备工作，包括设置优化算法、Loss 计算方法、评价指标计算方法：

优化器（optimizer）：即寻找最优解的方法，可计算和更新梯度，并根据梯度更新模型参数。飞桨框架在 paddle.optimizer 下提供了优化器相关 API。并且需要为优化器设置合适的学习率，或者指定合适的学习率策略，飞桨框架在 paddle.optimizer.lr 下提供了学习率策略相关的 API。

损失函数（loss）：用于评估模型的预测值和真实值的差距，模型训练过程即取得尽可能小的 loss 的过程。飞桨框架在 paddle.nn Loss层 提供了适用不同深度学习任务的损失函数相关 API。

评价指标（metrics）：用于评估模型的好坏，不同的任务通常有不同的评价指标。飞桨框架在 paddle.metric 下提供了评价指标相关 API。

使用 Model.fit 训练模型

做好模型训练的前期准备工作后，调用 Model.fit 接口来启动训练。 训练过程采用二层循环嵌套方式：内层循环完成整个数据集的一次遍历，采用分批次方式；外层循环根据设置的训练轮次完成数据集的多次遍历。因此需要指定至少三个关键参数：训练数据集，训练轮次和每批次大小:

训练数据集：传入之前定义好的训练数据集。

训练轮次（epoch）：训练时遍历数据集的次数，即外循环轮次。

批次大小（batch\_size）：内循环中每个批次的训练样本数。

使用 Model.evaluate 评估模型

训练好模型后，可在事先定义好的测试数据集上，使用 Model.evaluate 接口完成模型评估操作，结束后根据在 Model.prepare 中定义的 loss 和 metric 计算并返回相关评估结果。

返回格式是一个字典:

只包含loss， {'loss': xxx}

包含loss和一个评估指标， {'loss': xxx, 'metric name': xxx}

包含loss和多个评估指标， {'loss': xxx, 'metric name1': xxx, 'metric name2': xxx}

使用 Model.predict 执行推理

高层 API 中提供了 Model.predict 接口，可对训练好的模型进行推理验证。只需传入待执行推理验证的样本数据，即可计算并返回推理结果。

返回格式是一个列表：

模型是单一输出：[(numpy\_ndarray\_1, numpy\_ndarray\_2, …, numpy\_ndarray\_n)]

模型是多输出：[(numpy\_ndarray\_1, numpy\_ndarray\_2, …, numpy\_ndarray\_n), (numpy\_ndarray\_1, numpy\_ndarray\_2, …, numpy\_ndarray\_n), …]

如果模型是单一输出，则输出的形状为 [1, n]，n 表示数据集的样本数。其中每个 numpy\_ndarray\_n 是对应原始数据经过模型计算后得到的预测结果，类型为 numpy 数组，例如 mnist 分类任务中，每个 numpy\_ndarray\_n 是长度为 10 的 numpy 数组。

如果模型是多输出，则输出的形状为[m, n]，m 表示标签的种类数，在多标签分类任务中，m 会根据标签的数目而定

在模型训练过程中，通常会在如下场景中用到模型的保存与加载功能：

训练调优场景：

模型训练过程中定期保存模型，以便后续对不同时期的模型恢复训练或进行研究；

模型训练完毕，需要保存模型方便进行评估测试；

载入预训练模型，并对模型进行微调（fine-tune）。

推理部署场景：

模型训练完毕，在云、边、端不同的硬件环境中部署使用，飞桨提供了服务器端部署的 Paddle Inference、移动端/IoT端部署的 Paddle Lite、服务化部署的 Paddle Serving 等，以实现模型的快速部署上线。

模型迁移

从 PyTorch 迁移到飞桨 介绍如何将 PyTorch 训练代码迁移到飞桨。

CV - 快速上手 : 以 MobileNetV3 为例，介绍如何从 PyTorch 迁移到飞桨。

CV - 迁移经验总结 : 介绍 CV 各个方向从 PyTorch 迁移到飞桨的基本流程、常用工具、定位问题的思路及解决方法。

NLP - 快速上手 : 以 Bert 为例，介绍如何从 PyTorch 迁移到飞桨。

NLP - 迁移经验总结 : 介绍 NLP 各个方向从 PyTorch 迁移到飞桨的基本流程、常用工具、定位问题的思路及解决方法。

解读网络结构转换 : 介绍网络结构转换的思路和方法。

解读 Bert 模型权重转换 : 介绍如何进行不同框架下的模型权重转换。

PyTorch API 映射表 : 说明 PyTorch 1.8 版本与 Paddle 2.0 API 对应关系。

PyTorch 自定义算子转写教程 : 介绍 PyTorch 中自定义算子转写成 Paddle 自定义算子的思路和方法。

根据模型训练的常规流程，可将整个迁移任务划分为：模型前向对齐、数据读取对齐、评估指标对齐、损失函数对齐、反向梯度对齐、训练精度对齐。

模型前向对齐：PyTorch 的大部分 API 在飞桨中可找到对应 API，可以参考 PyTorch-PaddlePaddle API 映射表，模型组网部分代码直接进行手动转换即可；为了判断转换后的 飞桨模型组网能获得和 PyTorch 参考实现同样的输出，可将两个模型参数固定，并输入相同伪数据，观察两者的产出差异是否在阈值内。

数据读取对齐：相同的神经网络使用不同的数据训练和测试得到的结果往往会存在较大差异。为了能完全复现原始的模型，需要保证使用的数据完全相同，包括数据集的版本、使用的数据增强方式。

模型训练对齐：为了验证迁移后的模型能达到相同的精度，需要确保迁移模型使用的评价指标、损失函数与原模型相同，以便原模型与迁移后的模型对比。

评估指标对齐：飞桨提供了一系列 Metric 计算类，而 PyTorch 中目前可以通过组合的方式实现。应确保使用的指标与原代码含义一致，以便对照精度。

损失函数对齐：训练迁移后的模型时，需要使用与原代码相同的损失函数。飞桨与 PyTorch均提供了常用的损失函数，二者的 API 对应关系可参考 Loss 类 API 映射列表。

超参代码转换：训练过程中需要保证学习率、优化器、正则化系统等超参对齐。飞桨中的 optimizer 有 paddle.optimizer 等一系列实现，PyTorch 中则有 torch.optim 等一系列实现。对照 PaddlePaddle 正则化 API 文档与参考代码的优化器实现进行转换，用之后的反向梯度对齐统一验证该模块的正确性。

反向梯度对齐：前向对齐完成后，还需进行反向对齐，即确保迁移后的模型反向传播、权重更新与原模型一致。可以通过两轮训练进行检查，若迁移前后的模型第二轮训练的 loss 一致，则可以认为二者反向已对齐。

训练精度对齐：对比迁移前后模型的训练精度，若二者的差值在可以接受的误差范围内，则精度对齐完成。

训练性能对齐：在相同的硬件条件下，迁移前后的模型训练速度应接近。若二者差异非常大，则需要排查原因。

模型迁移本质上需要从模型训练与预测的角度去完成该任务，保证训练结果与预测结果与参考代码保持一致。

在模型预测过程中，模型组网、数据处理与加载、评估指标等均需要严格对齐，否则无法产出与参考代码完全相同的模型及预测结果。

在训练阶段，除了模型结构等元素，训练的损失函数、梯度、训练超参数、初始化方法以及训练精度也是我们需要迁移并对齐的内容。一个完整的模型训练包括定义模型结构并初始化，将处理后的数据送进网络，对输出的内容与真值计算损失函数，并反向传播与迭代的过程。

迁移准备：迁移工作首先需要安装必要的软件和工具（包括飞桨、PyTorch 或 TensorFlow 的安装、差异核验工具的安装等），然后准备要迁移的模型以及使用的数据集，同时了解源代码结构、跑通模型训练，最后对源代码进行解析，统计缺失算子。

模型前向对齐：这是迁移工作最基本的部分。在搭建神经网络时，会使用到框架提供的 API（内置的模块、函数等）。在迁移时，需要对这些 API 转换成飞桨对应的 API。转换前后的模型应具有相同的网络结构，使用相同的模型权重时，对于相同的输入，二者的输出结果应该一致。有时同一个神经网络有不同的版本、同一个版本有不同的实现方式或者在相同的神经网络下使用不同的超参，这些差别会对最终的收敛精度和性能造成一定影响。通常，我们以神经网络作者本身的实现为准，也可以参考不同框架（例如飞桨、TensorFlow、PyTorch 等）的官方实现或其他主流开源工具箱（例如 MMDetection）。PyTorch 的大部分 API 在飞桨中可找到对应 API，可以直接对模型组网部分代码涉及的 API 进行手动转换。为了判断转换后的飞桨模型组网能获得和 PyTorch 参考实现同样的输出，可将两个模型参数固定，并输入相同伪数据，观察两者的产出差异是否在阈值内。

小数据集数据读取对齐：数据读取对齐为了验证数据加载、数据预处理、数据增强与原始代码一致。为了快速验证数据读取对齐，建议准备一个小数据集（训练集和验证集各 8~16 张图像即可，压缩后数据大小建议在 20MB 以内）。

评估指标对齐：评估指标是模型精度的度量。在计算机视觉中，不同任务使用的评估指标有所不同（比如，在图像分类任务中，常用的指标是 Top-1 准确率与 Top-5 准确率；在图像语义分割任务中，常用的指标是 mIOU）为了检验迁移后的模型能否达到原模型的精度指标，需要保证使用的评估指标与原始代码一致以便对照。飞桨提供了一系列 Metric 计算类，而 PyTorch 中目前可以通过组合的方式实现或者调用第三方的 API。

损失函数对齐：损失函数是训练模型时的优化目标，使用的损失函数会影响模型的精度。模型迁移时，需要保证迁移后模型训练时使用的损失函数与原始代码中使用的损失函数一致，以便二者对照。飞桨与 PyTorch 均提供了常用的损失函数。

模型训练超参对齐：模型的训练超参包括学习率、优化器、正则化策略等。这些超参数指定了模型训练过程中网络参数的更新方式，训练超参数的设置会影响到模型的收敛速度及收敛精度。同样地，在模型迁移时，需要保证迁移前后模型使用的训练超参数一致，以便对照二者的收敛情况。飞桨中的优化器有 paddle.optimizer 等一系列实现，PyTorch 中则有 torch.optim 等一系列实现。完成超参对齐后，可以使用反向梯度对齐统一验证该模块的正确性。

反向梯度对齐：在完成前向对齐的基础上，还需进行反向梯度对齐。反向梯度对齐的目的是确保迁移后的模型反向传播以及权重更新的行为与原始模型一致，同时也是对上一步模型训练超参对齐的验证。具体的检验方法是通过两次（或以上）迭代训练进行检查，若迁移前后的模型第二轮训练的 loss 一致，则可以认为二者反向已对齐。

训练集数据读取对齐：相同的神经网络使用不同的数据训练和测试得到的结果往往会存在差异。因此，为了能复现原始代码的精度，需要保证使用的数据完全相同，包括数据集的版本、使用的数据预处理方法和流程、使用的数据增强方式等。

网络初始化对齐：对于不同的深度学习框架，网络初始化在大多情况下，即使值的分布完全一致，也无法保证值完全一致，这也是模型迁移不确定性比较大的地方。CNN 对于模型初始化相对不敏感，在迭代轮数与数据集足够的情况下，最终精度指标基本接近。而 transformer 系列模型、超分模型、领域自适应算法对于初始化比较敏感，需要对初始化进行重点检查。如果十分怀疑初始化导致的问题，建议将参考的初始化权重转成飞桨模型权重，加载该初始化模型训练，检查收敛精度。

训练精度对齐：模型训练的最终结果是为了得到一个精度达标的模型。不同的框架版本、是否为分布式训练等可能会对训练精度有影响，在迁移前需要分析清楚对标的框架、硬件等信息。对比迁移前后模型的训练精度，若二者的差值在可以接受的误差范围内，则精度对齐完成。同时，如果在相同的硬件条件下，迁移前后的模型训练速度应接近。若二者差异非常大，则需要排查原因。

模型预测验证：模型训练完成之后，需要使用测试集对该模型基于训练引擎进行预测，确认预测结果与实际一致。 其中，2~5 是迁移的重点，其他模块比如反向梯度、优化器、学习率生成等，要么本身结构单一，要么依赖已开发完成的网络结果才能和对标脚本形成对比，这些模块的脚本开发难度较小

在飞桨中，paddle.nn 提供了常见神经网络层的实现，如卷积网络相关的 Conv1D、Conv2D、Conv3D，循环神经网络相关的 RNN、LSTM、GRU 等，基于这些 API 可完成大多数神经网络的搭建，详细清单可在 API 文档 中查看。同时，飞桨提供继承类（class）的方式构建网络，paddle.nn.Layer 类是构建所有网络的基类，通过构建一个继承基类的子类，并在子类中添加层（layer，如卷积层、全连接层等）即可实现网络的构建。飞桨中的组网 API 与 PyTorch 的 API 较为相似，部分 API 在参数名或功能上存在一定差异。部分 API 在完整的飞桨与 PyTorch 组网类 API 映射表请参考组网类 API 映射列表。

使用飞桨构建网络的步骤如下：

创建一个继承自 paddle.nn.Layer 的类；

在类的构造函数 \_\_init\_\_ 中定义组网用到的神经网络层（layer）；

在类的前向计算函数 forward 中使用定义好的 layer 执行前向计算。

基于 PyTorch 的网络结构

PyTorch 模块通常继承torch.nn.Module，对模块的定义方法与飞桨类似，即在 \_\_init\_\_中定义模块中用到的子模块，然后 forward函数中定义前向传播的方式。

使用 PyTorch 构建网络的步骤如下：

创建一个继承自 torch.nn.Module 的类。

在类的构造函数 \_\_init\_\_ 中定义组网用到的神经网络层（layer）。

在类的前向计算函数 forward 中使用定义好的 layer 执行前向计算。

通过阅读源代码，可以发现 MobileNetV3 的基本 block 是在 InvertedResidual 类中定义的，在 block 的搭建中又使用到了 Conv2dNormActivation 与 SqueezeExcitation 这两个模块（这两个模块是从 ..ops.misc 中 import 的，因此需要到 ../ops/misc.py 中查看这两个模块的实现；代码转换时，这两个模块也需一并转换）。MobileNetV3 类则使用前面定义好的 block 来搭建完整的网络。由此，我们可以知道，需要转换的模块（类）有 Conv2dNormActivation, SqueezeExcitation, InvertedResidual 以及 MobileNetV3。

在做下游任务模型（如检测、分割）转换时，需要留意原始代码是否使用了预训练模型。若使用了预训练模型，则需要将 PyTorch 的预训练权重转换为飞桨格式的模型权重。

此外，分析论文是否能通过飞桨已有模型或者少量修改即可实现。模型的实现具有相通性，迁移过程中可参考和借鉴飞桨已实现模型的代码，这样可以减少模型开发的工作量，提升模型迁移效率。

如果作者使用 PyTorch 基础 API 进行网络结构设计，由于 PyTorch 的 API 和飞桨的 API 非常相似，可以将调用的 PyTorch API 手工替换成对应的飞桨 API 。

手工转换 API 工作，可以参考PyTorch-飞桨 API 映射表，将原始代码中调用的 PyTorch API（即原始代码中 import 的 torch 包的类、函数，例如 torch.nn 中的模块及 torch.nn.functional 中的函数等）替换成相应的飞桨 API。

需要注意的是，飞桨的部分 API 与 PyTorch 中对应的 API 在功能与参数上存在一定区别，转换时需要留意。

模型前向对齐

前向精度的对齐十分简单，我们只需要保证两者输入是一致的前提下，观察得到的输出是否一致。模型在前向对齐验证时，需要调用model.eval()方法，以消除随机因素的影响，比如 BatchNorm、Dropout 等。

下面以 MobileNetV3 为例，说明验证模型前向对齐的步骤。

获取 PyTorch 模型权重，保存为 mobilenet\_v3\_small.pth，并将 PyTorch 权重转换为飞桨权重，保存为 mv3\_small\_paddle.pdparams。这是为了前向对齐时保证两个网络加载的权重相同。对于权重转换的详细介绍，请参考：迁移经验汇总。

为了方便快速验证，可以根据模型的输入尺寸，利用 numpy 生成随机伪数据（包括输入图片及标签），并保存为 fake\_data.npy 和 fake\_label.npy。

将生成的伪数据 (fake\_data.npy 和 fake\_label.npy)，送入 PyTorch 模型获取输出，使用 reprod\_log 保存结果。

PyTorch 和 Paddle 都是通过序列化和反序列化模型的 state dict （状态字典）来进行参数权重的存储和加载的。state dict 从数据结构上来看就是一个字典（比如 Python 中的 dict），其中 key 是模型参数的名称（数据类型为 string），而 value 则为 key 所对应的值（数据类型为 Tensor）。参数存储时，先获取目标对象的 state dict ，然后将 state dict 存储至磁盘；参数载入时，先从磁盘载入保存的 state dict ，然后通过 set\_state\_dict() 方法配置到目标对象中。

利用 state dict 进行权重格式转换

了解了模型的存储和加载以及相关的 state dict 之后，来看一下模型格式的转换的具体步骤。一般来说，可以通过 state dict 的相互转换来帮助进行模型格式的转换。

以从 PyTorch 框架到 Paddle 框架的模型权重转换为例，转换的具体流程为：

加载 PyTorch 模型得到 state dict

PyTorch 下的 state dict 转换为 Paddle 下的 state dict

保存 Paddle 下的 state dict 得到 Paddle 模型

在深度学习模型构建上，飞桨框架支持动态图编程和静态图编程两种方式，其代码编写和执行方式均存在差异。

动态图编程： 采用 Python 的编程风格，解析式地执行每一行网络代码，并同时返回计算结果。在 模型开发 章节中，介绍的主要是动态图编程方式。

静态图编程： 采用先编译后执行的方式。需先在代码中预定义完整的神经网络结构，飞桨框架会将神经网络描述为 Program 的数据结构，并对 Program 进行编译优化，再调用执行器获得计算结果。

飞桨框架在设计时，考虑同时兼顾动态图的高易用性和静态图的高性能优势，采用动静统一的方案：

在模型开发和训练时，推荐采用动态图编程。 可获得更好的编程体验、更易用的接口、更友好的调试交互机制。

在模型推理部署时，推荐采用动态图转静态图（以下简称：动转静）。平滑衔接将训好的动态图模型自动保存为静态图模型，可获得更好的模型运行性能

使用 paddle.jit.save 保存模型，通常是在后台执行了两个步骤：

先执行了动转静。当然如果前面已经执行了动转静训练，则跳过这一步。在处理逻辑上，主要包含两个主要模块：

模型结构层面：将动态图模型中被 @paddle.jit.to\_static 装饰的函数转化为完整的静态图 Program。

模型参数层面：将动态图模型中的参数（Parameters 和 Buffers ）转为 Persistable=True 的静态图模型参数 Variable。

再将静态图模型和参数导出为磁盘文件。Program 和 Variable 都可以直接序列化导出为磁盘文件，与前端代码完全解耦，导出的文件包括：

后缀为 .pdmodel 的模型结构文件；

后缀为 .pdiparams 的模型参数文件；

后缀为 .pdiparams.info 的和参数状态有关的额外信息文件。

类似的，使用 paddle.jit.load 加载模型，即将上述三个文件加载为静态图模型的 Program 和 Variable，可用于执行静态图模式下训练调优或验证推理效果。

无论是动态图训练，还是动转静训练，都支持通过 paddle.jit.save 自动转为静态图模型，只是使用过程中有一些配置差异

动态图模型训练完成后，保存为静态图模型用于推理部署，主要包括三个步骤：

切换 eval() 模式： 类似 Dropout 、LayerNorm 等接口在 train() 和 eval() 的行为存在较大的差异，在模型导出前，请务必确认模型已切换到正确的模式，否则导出的模型在预测阶段可能出现输出结果不符合预期的情况。用于推理部署切换到 eval()模式，用于后续训练调优则切换到 train()模式。

构造 InputSpec 信息： InputSpec 用于表示模型输入数据的 shape、dtype、name 信息，是辅助动静转换的必要描述信息。这是由于静态图模型在调用执行器前并不执行实际操作，因此也并不读入实际数据，需要设置 “占位符” 表示输入数据。详细请参见 InputSpec 的用法介绍 。

调用 save 接口： 调用 paddle.jit.save 接口，若传入的参数是类实例（如示例中的 Layer 类），则框架会默认对神经网络中定义的前向计算 forward 函数进行 @paddle.jit.to\_static 装饰（执行动转静），并导出其对应的模型文件和参数文件。

变量在不同分支类型须保持一致

模型代码中的 if...else 语句在动转静之后，会被转换成统一的范式。当依赖的条件变量（如下样例中的 x > y ）是一个 Tensor 类型时，if...else 分支中所有的变量类型须保持一致。当类型不一致时，后续的类型检查将抛出异常

张量在不同分支 shape 须保持一致

依赖控制流的 if...else 语句在动转静生成中间表示 Program 时，要求两个分支中同名张量的 shape 必须保持一致，因为静态图下会对两个分支的输出进行动态 select input 操作，故须保证无论条件变量 x > y 取何值，选取的张量 shape 都是一致的。否则在后续组网或者训练时，出现因 shape 不同而报错

条件变量类型须保持不变

While 的条件变量在循环过程中的类型应保持不变，因为循环变量的类型将会决定其是保持 Python 语法运行，或是转为飞桨的 while\_loop API。保持条件变量类型不变才能确保模型正确地被动转静。

飞桨推理产品 paddle inference 和主框架的 Model.predict 均可实现推理预测，Paddle Inference 是飞桨的原生推理库， 作用于服务器端和云端，提供高性能的推理能力，主框架的 Model 对象是一个具备训练、测试、推理的神经网络。相比于 Model.predict，inference 可使用 MKLDNN、CUDNN、TensorRT 进行预测加速，同时支持用 X2Paddle 工具从第三方框架（TensorFlow、Pytorh 、 Caffe 等）产出的模型，可联动 PaddleSlim，支持加载量化、裁剪和蒸馏后的模型部署。Model.predict 适用于训练好的模型直接进行预测，paddle inference 适用于对推理性能、通用性有要求的用户，针对不同平台不同的应用场景进行了深度的适配优化，保证模型在服务器端即训即用，快速部署

高性能实现

内存/显存复用提升服务吞吐量

在推理初始化阶段，对模型中的 OP 输出 Tensor 进行依赖分析，将两两互不依赖的 Tensor 在内存/显存空间上进行复用，进而增大计算并行量，提升服务吞吐量。

细粒度 OP 横向纵向融合减少计算量

在推理初始化阶段，按照已有的融合模式将模型中的多个 OP 融合成一个 OP，减少了模型的计算量的同时，也减少了 Kernel Launch 的次数，从而能提升推理性能。目前 Paddle Inference 支持的融合模式多达几十个。

内置高性能的 CPU/GPU Kernel

内置同 Intel、Nvidia 共同打造的高性能 kernel，保证了模型推理高性能的执行。

多功能集成

集成 TensorRT 加快 GPU 推理速度

Paddle Inference 采用子图的形式集成 TensorRT，针对 GPU 推理场景，TensorRT 可对一些子图进行优化，包括 OP 的横向和纵向融合，过滤冗余的 OP，并为 OP 自动选择最优的 kernel，加快推理速度。

集成 oneDNN CPU 推理加速引擎

一行代码开始 oneDNN 加速，快捷高效。

支持 PaddleSlim 量化压缩模型的部署

PaddleSlim 是飞桨深度学习模型压缩工具，Paddle Inference 可联动 PaddleSlim，支持加载量化、裁剪和蒸馏后的模型并部署，由此减小模型存储空间、减少计算占用内存、加快模型推理速度。其中在模型量化方面，Paddle Inference 在 X86 CPU 上做了深度优化，常见分类模型的单线程性能可提升近 3 倍，ERNIE 模型的单线程性能可提升 2.68 倍。

支持 X2Paddle 转换得到的模型

除支持飞桨训练的模型外，也支持用 X2Paddle 工具从第三方框架（比如 TensorFlow、PyTorch 或者 Caffe 等）产出的模型