PaddlePaddle

PaddlePaddle

2021 年 02 月 26 日

Contents

1 整体介绍 **3**

1.1 基本概念 . . . 3
1.2 升级指南 . . . 16
1.3 版本迁移工具 . . . 28

2 模型开发 **33**

2.1 10 分钟快速上手飞桨（PaddlePaddle） . . . 34
2.2 数据集定义与加载 . . . 36
2.3 数据预处理 . . . 39
2.4 模型组网 . . . 41
2.5 训练与预测 . . . 44
2.6 资源配置 . . . 49
2.7 自定义指标 . . . 53
2.8 模型存储与载入 . . . 59
2.9 模型导出 ONNX 协议 . . . 77

3 VisualDL 工具 **81**

3.1 VisualDL 工具简介 . . . 81
3.2 VisualDL 使用指南 . . . 86

4 动态图转静态图 **97**

4.1 基本用法 . . . 98
4.2 内部架构原理 . . . 101
4.3 支持语法列表 . . . 102
4.4 InputSpec 功能介绍 . . . 105
4.5 报错信息处理 . . . 111
4.6 调试方法 . . . 114

5 推理部署 **121**
5.1 飞桨推理产品简介 121

6 分布式训练 **185**
6.1 分布式训练快速开始 185
6.2 使用 FleetAPI 进行分布式训练 188

7 昆仑 XPU 芯片运行飞桨 **195**
7.1 飞桨对昆仑 XPU 芯片的支持 195
7.2 飞桨框架昆仑 XPU 版安装说明 196
7.3 飞桨框架昆仑 XPU 版训练示例 200

8 自定义 OP **201**
8.1 如何写新的 C++ OP 201
8.2 C++ OP 相关注意事项 218
8.3 如何写新的 Python OP 228
8.4 如何在框架外部自定义 C++ OP 231

9 参与开发 **241**
9.1 本地开发指南 241
9.2 提交 PR 注意事项 245
9.3 FAQ 247

10 其他说明 **249**
10.1 硬件支持 249
10.2 飞桨框架 API 映射表 250

飞桨开源框架 (PaddlePaddle) 是一个易用、高效、灵活、可扩展的深度学习框架。

你可参考飞桨框架的 Github 了解详情，也可阅读 版本说明 了解 2.0 版本的特性。

使用教程分为如下的模块：

- 整体介绍：飞桨框架 2.0 新特性的介绍与飞桨框架 2.0 升级指南的说明。
- 模型开发：飞桨框架 2.0 模型开发全流程说明。
- 模型可视化：介绍如何用 VisualDL 实现飞桨框架模型的可视化。
- 动态图转静态图：介绍飞桨框架动态图转静态图的方法。
- 预测部署：介绍如何使用训练好的模型进行预测。
- 分布式训练：介绍如何使用分布式进行训练。
- 昆仑 XPU 芯片运行飞桨：介绍如何在昆仑 XPU 芯片环境上安装和使用飞桨。
- 自定义 OP：介绍飞桨框架自定义 OP 的方法。
- 参与开发：介绍如何参与飞桨框架的开发。
- 其他说明：飞桨框架的其他说明文档。

CHAPTER 1

整体介绍

您可以通过下面的内容，了解更多飞桨框架 2.0 的内容:

- 基本概念 : 飞桨框架 2.0 基本概念的介绍。
- 升级指南: 介绍飞桨框架 2.0 的主要变化和如何升级。
- 版本迁移工具: 介绍飞桨框架版本转换工具的使用。

1.1 基本概念

让我们从学习飞桨的基本概念这里开始:

- Tensor 概念介绍 : 飞桨中数据的表示方式，Tensor 概念介绍。
- 飞桨广播介绍 : 飞桨中广播概念的介绍。

1.1.1 Tensor 概念介绍

飞桨（PaddlePaddle，以下简称 Paddle）和其他深度学习框架一样，使用 **Tensor** 来表示数据，在神经网络中传递的数据均为 **Tensor**。

Tensor 可以将其理解为多维数组，其可以具有任意多的维度，不同 **Tensor** 可以有不同的**数据类型** (dtype) 和**形状** (shape)。

同一 **Tensor** 的中所有元素的 dtype 均相同。如果你对 Numpy 熟悉，**Tensor** 是类似于 **Numpy array** 的概念。

Tensor 的创建

首先，让我们开始创建一个 **Tensor** , 并用 **ndim** 表示 **Tensor** 维度的数量:

1. 创建类似于 vector 的 1-D Tensor，其 ndim 为 1

```
# 可通过 dtype 来指定 Tensor 数据类型，否则会创建 float32 类型的 Tensor
ndim_1_tensor = paddle.to_tensor([2.0, 3.0, 4.0], dtype='float64')
print(ndim_1_tensor)
```

```
Tensor(shape=[3], dtype=float64, place=CUDAPlace(0), stop_gradient=True,
       [2., 3., 4.])
```

特殊地，如果仅输入单个 scalar 类型数据（例如 float/int/bool 类型的单个元素），则会创建 shape 为 [1] 的 **Tensor**

```
paddle.to_tensor(2)
paddle.to_tensor([2])
```

上述两种创建方式完全一致，shape 均为 [1]，输出如下：

```
Tensor(shape=[1], dtype=int64, place=CUDAPlace(0), stop_gradient=True,
       [2])
```

2. 创建类似于 matrix 的 2-D Tensor，其 ndim 为 2

```
ndim_2_tensor = paddle.to_tensor([[1.0, 2.0, 3.0],
                                  [4.0, 5.0, 6.0]])
print(ndim_2_tensor)
```

```
Tensor(shape=[2, 3], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
       [[1., 2., 3.],
        [4., 5., 6.]])
```

3. 同样地，还可以创建 ndim 为 3、4…N 等更复杂的多维 Tensor

```
# Tensor 可以有任意数量的轴（也称为维度）
ndim_3_tensor = paddle.to_tensor([[[1, 2, 3, 4, 5],
                                   [6, 7, 8, 9, 10]],
                                  [[11, 12, 13, 14, 15],
                                   [16, 17, 18, 19, 20]]])
print(ndim_3_tensor)
```

```
Tensor(shape=[2, 2, 5], dtype=int64, place=CUDAPlace(0), stop_gradient=True,
       [[[1, 2, 3, 4, 5],
         [ 6,  7,  8,  9, 10]],

        [[11, 12, 13, 14, 15],
         [16, 17, 18, 19, 20]]])
```

上述不同 ndim 的 **Tensor** 可以可视化的表示为：

你可以通过 Tensor.numpy() 方法方便地将 **Tensor** 转化为 **Numpy array**：

```
ndim_2_tensor.numpy()
```

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

Tensor 不仅支持 floats、ints 类型数据，也支持 complex numbers 数据：

```
ndim_2_complex_tensor = paddle.to_tensor([[1+1j, 2+2j],
                                          [3+3j, 4+4j]])
```

如果输入为复数数据，则 **Tensor** 的 dtype 为 `complex64` 或 `complex128`，其每个元素均为 1 个复数：

```
Tensor(shape=[2, 2], dtype=complex64, place=CUDAPlace(0), stop_gradient=True,
       [[(1+1j), (2+2j)],
        [(3+3j), (4+4j)]])
```

Tensor 必须形状规则，类似于“矩形”的概念，也就是，沿任何一个轴（也称作维度）上，元素的数量都是相等的，如果为以下情况：

```
ndim_2_tensor = paddle.to_tensor([[1.0, 2.0],
                                  [4.0, 5.0, 6.0]])
```

该情况下将会抛出异常：

```
ValueError:
    Faild to convert input data to a regular ndarray :
     - Usually this means the input data contains nested lists with different lengths.
```

上面介绍了通过 Python 数据来创建 **Tensor** 的方法，我们也可以通过 **Numpy array** 来创建 **Tensor**：

```
ndim_1_tensor = paddle.to_tensor(numpy.array([1.0, 2.0]))

ndim_2_tensor = paddle.to_tensor(numpy.array([[1.0, 2.0],
                                              [3.0, 4.0]]))
```

(下页继续)

(续上页)

```
ndim_3_tensor = paddle.to_tensor(numpy.random.rand(3, 2))
```

创建的 **Tensor** 与原 **Numpy array** 具有相同的 shape 与 dtype。

如果要创建一个指定 shape 的 **Tensor**，Paddle 也提供了一些 API：

```
paddle.zeros([m, n])              # 创建数据全为 0，shape 为 [m, n] 的 Tensor
paddle.ones([m, n])               # 创建数据全为 1，shape 为 [m, n] 的 Tensor
paddle.full([m, n], 10)           # 创建数据全为 10，shape 为 [m, n] 的 Tensor
paddle.arange(start, end, step)   # 创建从 start 到 end，步长为 step 的 Tensor
paddle.linspace(start, end, num) # 创建从 start 到 end，元素个数固定为 num 的 Tensor
```

Tensor 的 shape

基本概念

查看一个 **Tensor** 的形状可以通过 **Tensor.shape**，shape 是 **Tensor** 的一个重要属性，以下为相关概念：

1. shape：描述了 tensor 的每个维度上元素的数量
2. ndim：tensor 的维度数量，例如 vector 的 ndim 为 1，matrix 的 ndim 为 2.
3. axis 或者 dimension：指 tensor 某个特定的维度
4. size：指 tensor 中全部元素的个数

让我们来创建 1 个 4-D **Tensor**，并通过图形来直观表达以上几个概念之间的关系；

```
ndim_4_tensor = paddle.ones([2, 3, 4, 5])
```

```
print("Data Type of every element:", ndim_4_tensor.dtype)
print("Number of dimensions:", ndim_4_tensor.ndim)
print("Shape of tensor:", ndim_4_tensor.shape)
print("Elements number along axis 0 of tensor:", ndim_4_tensor.shape[0])
print("Elements number along the last axis of tensor:", ndim_4_tensor.shape[-1])
```

```
Data Type of every element: VarType.FP32
Number of dimensions: 4
Shape of tensor: [2, 3, 4, 5]
Elements number along axis 0 of tensor: 2
Elements number along the last axis of tensor: 5
```

对 shape 进行操作

重新定义 **Tensor** 的 shape 在实际编程中具有重要意义。

```
ndim_3_tensor = paddle.to_tensor([[[1, 2, 3, 4, 5],
                                   [6, 7, 8, 9, 10]],
                                  [[11, 12, 13, 14, 15],
                                   [16, 17, 18, 19, 20]],
                                  [[21, 22, 23, 24, 25],
                                   [26, 27, 28, 29, 30]]])
print("the shape of ndim_3_tensor:", ndim_3_tensor.shape)
```

```
the shape of ndim_3_tensor: [3, 2, 5]
```

Paddle 提供了 reshape 接口来改变 Tensor 的 shape：

```
ndim_3_tensor = paddle.reshape(ndim_3_tensor, [2, 5, 3])
print("After reshape:", ndim_3_tensor.shape)
```

```
After reshape: [2, 5, 3]
```

在指定新的 shape 时存在一些技巧：

1. -1 表示这个维度的值是从 Tensor 的元素总数和剩余维度推断出来的。因此，有且只有一个维度可以被设置为-1。**2.** 0 表示实际的维数是从 Tensor 的对应维数中复制出来的，因此 shape 中 0 的索引值不能超过 x 的维度。

有一些例子可以很好解释这些技巧：

```
origin:[3, 2, 5] reshape:[3, 10]      actual: [3, 10]
origin:[3, 2, 5] reshape:[-1]         actual: [30]
origin:[3, 2, 5] reshape:[0, 5, -1]   actual: [3, 5, 2]
```

可以发现，reshape 为 [-1] 时，会将 tensor 按其在计算机上的内存分布展平为 1-D Tensor。

```
print("Tensor flattened to Vector:", paddle.reshape(ndim_3_tensor, [-1]).numpy())
```

```
Tensor flattened to Vector: [1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19␣
↪20 21 22 23 24 25 26 27 28 29 30]
```

Tensor 其他属性

Tensor 的 dtype

Tensor 的数据类型，可以通过 Tensor.dtype 来查看，dtype 支持：’ bool’ , ’ float16’ , ’ float32’ , ’ float64’ , ’ uint8’ , ’ int8’ , ’ int16’ , ’ int32’ , ’ int64’。

- 通过 Python 元素创建的 Tensor，可以通过 dtype 来进行指定，如果未指定：
  - 对于 python 整型数据，则会创建 int64 型 Tensor
  - 对于 python 浮点型数据，默认会创建 float32 型 Tensor，并且可以通过 set_default_type 来调整浮点型数据的默认类型。
- 通过 Numpy array 创建的 Tensor，则与其原来的 dtype 保持相同。

```
print("Tensor dtype from Python integers:", paddle.to_tensor(1).dtype)
print("Tensor dtype from Python floating point:", paddle.to_tensor(1.0).dtype)
```

```
Tensor dtype from Python integers: VarType.INT64
Tensor dtype from Python floating point: VarType.FP32
```

Paddle 提供了 **cast** 接口来改变 dtype：

```
float32_tensor = paddle.to_tensor(1.0)

float64_tensor = paddle.cast(float32_tensor, dtype='float64')
print("Tensor after cast to float64:", float64_tensor.dtype)

int64_tensor = paddle.cast(float32_tensor, dtype='int64')
print("Tensor after cast to int64:", int64_tensor.dtype)
```

```
Tensor after cast to float64: VarType.FP64
Tensor after cast to int64: VarType.INT64
```

Tensor 的 place

初始化 **Tensor** 时可以通过 **place** 来指定其分配的设备位置，可支持的设备位置有三种：CPU/GPU/固定内存，其中固定内存也称为不可分页内存或锁页内存，其与 GPU 之间具有更高的读写效率，并且支持异步传输，这对网络整体性能会有进一步提升，但其缺点是分配空间过多时可能会降低主机系统的性能，因为其减少了用于存储虚拟内存数据的可分页内存。

- **创建 CPU 上的 Tensor**：

```
cpu_tensor = paddle.to_tensor(1, place=paddle.CPUPlace())
print(cpu_tensor)
```

```
Tensor(shape=[1], dtype=int64, place=CPUPlace, stop_gradient=True,
       [1])
```

- **创建 GPU 上的 Tensor**：

```
gpu_tensor = paddle.to_tensor(1, place=paddle.CUDAPlace(0))
print(gpu_tensor)
```

```
Tensor(shape=[1], dtype=int64, place=CUDAPlace(0), stop_gradient=True,
       [1])
```

- **创建固定内存上的 Tensor**：

```
pin_memory_tensor = paddle.to_tensor(1, place=paddle.CUDAPinnedPlace())
print(pin_memory_tensor)
```

```
Tensor(shape=[1], dtype=int64, place=CUDAPinnedPlace, stop_gradient=True,
       [1])
```

Tensor 的 name

Tensor 的 name 是其唯一的标识符，为 python 字符串类型，查看一个 Tensor 的 name 可以通过 Tensor.name 属性。默认地，在每个 Tensor 创建时，Paddle 会自定义一个独一无二的 name。

```
print("Tensor name:", paddle.to_tensor(1).name)
```

```
Tensor name: generated_tensor_0
```

Tensor 的操作

索引和切片

您可以通过索引或切片方便地访问或修改 Tensor。Paddle 使用标准的 Python 索引规则与 Numpy 索引规则，与 Indexing a list or a string in Python类似。具有以下特点：

1. 基于 0-n 的下标进行索引，如果下标为负数，则从尾部开始计算
2. 通过冒号 `:` 分隔切片参数 `start:stop:step` 来进行切片操作，其中 start、stop、step 均可缺省

访问 Tensor

- 针对 1-D **Tensor**，则仅有单个轴上的索引或切片：

```
ndim_1_tensor = paddle.to_tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
print("Origin Tensor:", ndim_1_tensor.numpy())

print("First element:", ndim_1_tensor[0].numpy())
print("Last element:", ndim_1_tensor[-1].numpy())
print("All element:", ndim_1_tensor[:].numpy())
print("Before 3:", ndim_1_tensor[:3].numpy())
print("From 6 to the end:", ndim_1_tensor[6:].numpy())
print("From 3 to 6:", ndim_1_tensor[3:6].numpy())
print("Interval of 3:", ndim_1_tensor[::3].numpy())
print("Reverse:", ndim_1_tensor[::-1].numpy())
```

```
Origin Tensor: [0 1 2 3 4 5 6 7 8])
First element: [0]
Last element: [8]
All element: [0 1 2 3 4 5 6 7 8]
Before 3: [0 1 2]
From 6 to the end: [6 7 8]
From 3 to 6: [3 4 5]
Interval of 3: [0 3 6]
Reverse: [8 7 6 5 4 3 2 1 0]
```

- 针对 2-D 及以上的 **Tensor**，则会有多个轴上的索引或切片：

```
ndim_2_tensor = paddle.to_tensor([[0, 1, 2, 3],
                                  [4, 5, 6, 7],
                                  [8, 9, 10, 11]])
print("Origin Tensor:", ndim_2_tensor.numpy())
print("First row:", ndim_2_tensor[0].numpy())
print("First row:", ndim_2_tensor[0, :].numpy())
print("First column:", ndim_2_tensor[:, 0].numpy())
print("Last column:", ndim_2_tensor[:, -1].numpy())
print("All element:", ndim_2_tensor[:].numpy())
print("First row and second column:", ndim_2_tensor[0, 1].numpy())
```

```
Origin Tensor: [[ 0  1  2  3]
                [ 4  5  6  7]
                [ 8  9 10 11]]
First row: [0 1 2 3]
First row: [0 1 2 3]
```

(下页继续)

(续上页)

```
First column: [0 4 8]
Last column: [ 3  7 11]
All element: [[ 0  1  2  3]
              [ 4  5  6  7]
              [ 8  9 10 11]]
First row and second column: [1]
```

索引或切片的第一个值对应 axis 0，第二个值对应 axis 1，以此类推，如果某个 axis 上未指定索引，则默认为 : 。例如：

```
ndim_2_tensor[1]
ndim_2_tensor[1, :]
```

这两种操作的结果是完全相同的。

```
Tensor(shape=[4], dtype=int64, place=CPUPlace, stop_gradient=True,
       [4, 5, 6, 7])
```

修改 Tensor

注意：

请慎重通过索引或切片修改 Tensor，该操作会**原地**修改该 Tensor 的数值，且原值不会被保存。如果被修改的 Tensor 参与梯度计算，将仅会使用修改后的数值，这可能会给梯度计算引入风险。Paddle 之后将会对具有风险的操作进行检测和报错。

与访问 Tensor 类似，修改 Tensor 可以在单个或多个轴上通过索引或切片操作。同时，支持将多种类型的数据赋值给该 Tensor，当前支持的数据类型有：`int`, `float`, `numpy.ndarray`, `Tensor`。

```
import paddle
import numpy as np

x = paddle.to_tensor(np.ones((2, 3)).astype(np.float32)) # [[1., 1., 1.], [1., 1., 1.
↪]]

x[0] = 0                      # x : [[0., 0., 0.], [1., 1., 1.]]        id(x) =␣
↪4433705584
x[0:1] = 2.1                  # x : [[2.1, 2.1, 2.1], [1., 1., 1.]]     id(x) =␣
↪4433705584
x[...] = 3                    # x : [[3., 3., 3.], [3., 3., 3.]]        id(x) =␣
↪4433705584

x[0:1] = np.array([1,2,3])    # x : [[1., 2., 3.], [3., 3., 3.]]        id(x) =␣
↪4433705584
```

(下页继续)

(续上页)

```
x[1] = paddle.ones([3])        # x : [[1., 2., 3.], [1., 1., 1.]]        id(x) =␣
↪4433705584
```

同时，Paddle 还提供了丰富的 Tensor 操作的 API，包括数学运算符、逻辑运算符、线性代数相关等 100 余种 API，这些 API 调用有两种方法：

```
x = paddle.to_tensor([[1.1, 2.2], [3.3, 4.4]], dtype="float64")
y = paddle.to_tensor([[5.5, 6.6], [7.7, 8.8]], dtype="float64")

print(paddle.add(x, y), "\n")
print(x.add(y), "\n")
```

```
Tensor(shape=[2, 2], dtype=float64, place=CUDAPlace(0), stop_gradient=True,
       [[6.60000000, 8.80000000],
        [        11., 13.20000000]])

Tensor(shape=[2, 2], dtype=float64, place=CUDAPlace(0), stop_gradient=True,
       [[6.60000000, 8.80000000],
        [        11., 13.20000000]])
```

可以看出，使用 **Tensor 类成员函数**和 **Paddle API** 具有相同的效果，由于 **类成员函数**操作更为方便，以下均从 **Tensor 类成员函数**的角度，对常用 **Tensor** 操作进行介绍。

数学运算符

```
x.abs()                         # 逐元素取绝对值
x.ceil()                        # 逐元素向上取整
x.floor()                       # 逐元素向下取整
x.round()                       # 逐元素四舍五入
x.exp()                         # 逐元素计算自然常数为底的指数
x.log()                         # 逐元素计算 x 的自然对数
x.reciprocal()                  # 逐元素求倒数
x.square()                      # 逐元素计算平方
x.sqrt()                        # 逐元素计算平方根
x.sin()                         # 逐元素计算正弦
x.cos()                         # 逐元素计算余弦
x.add(y)                        # 逐元素相加
x.subtract(y)                   # 逐元素相减
x.multiply(y)                   # 逐元素相乘
x.divide(y)                     # 逐元素相除
```

(下页继续)

(续上页)

```
x.mod(y)                        # 逐元素相除并取余
x.pow(y)                        # 逐元素幂运算
x.max()                         # 指定维度上元素最大值，默认为全部维度
x.min()                         # 指定维度上元素最小值，默认为全部维度
x.prod()                        # 指定维度上元素累乘，默认为全部维度
x.sum()                         # 指定维度上元素的和，默认为全部维度
```

Paddle 对 python 数学运算相关的魔法函数进行了重写，以下操作与上述结果相同。

```
x + y   -> x.add(y)             # 逐元素相加
x - y   -> x.subtract(y)        # 逐元素相减
x * y   -> x.multiply(y)        # 逐元素相乘
x / y   -> x.divide(y)          # 逐元素相除
x % y   -> x.mod(y)             # 逐元素相除并取余
x ** y -> x.pow(y)              # 逐元素幂运算
```

逻辑运算符

```
x.isfinite()                    # 判断 tensor 中元素是否是有限的数字，即不包括 inf 与 nan
x.equal_all(y)                  # 判断两个 tensor 的全部元素是否相等，并返回 shape 为 [1] 的
bool Tensor
x.equal(y)                      # 判断两个 tensor 的每个元素是否相等，并返回 shape 相同的 bool␣
↪Tensor
x.not_equal(y)                  # 判断两个 tensor 的每个元素是否不相等
x.less_than(y)                  # 判断 tensor x 的元素是否小于 tensor y 的对应元素
x.less_equal(y)                 # 判断 tensor x 的元素是否小于或等于 tensor y 的对应元素
x.greater_than(y)               # 判断 tensor x 的元素是否大于 tensor y 的对应元素
x.greater_equal(y)              # 判断 tensor x 的元素是否大于或等于 tensor y 的对应元素
x.allclose(y)                   # 判断 tensor x 的全部元素是否与 tensor y 的全部元素接近，并返回
shape 为 [1] 的 bool Tensor
```

同样地，Paddle 对 python 逻辑比较相关的魔法函数进行了重写，以下操作与上述结果相同。

```
x == y   -> x.equal(y)          # 判断两个 tensor 的每个元素是否相等
x != y   -> x.not_equal(y)      # 判断两个 tensor 的每个元素是否不相等
x < y    -> x.less_than(y)      # 判断 tensor x 的元素是否小于 tensor y 的对应元素
x <= y   -> x.less_equal(y)     # 判断 tensor x 的元素是否小于或等于 tensor y 的对应元素
x > y    -> x.greater_than(y)   # 判断 tensor x 的元素是否大于 tensor y 的对应元素
x >= y   -> x.greater_equal(y) # 判断 tensor x 的元素是否大于或等于 tensor y 的对应元素
```

以下操作仅针对 bool 型 Tensor：

```
x.logical_and(y)                # 对两个 bool 型 tensor 逐元素进行逻辑与操作
x.logical_or(y)                 # 对两个 bool 型 tensor 逐元素进行逻辑或操作
x.logical_xor(y)                # 对两个 bool 型 tensor 逐元素进行逻辑亦或操作
x.logical_not(y)                # 对两个 bool 型 tensor 逐元素进行逻辑非操作
```

线性代数相关

```
x.cholesky()                    # 矩阵的 cholesky 分解
x.t()                           # 矩阵转置
x.transpose([1, 0])             # 交换 axis 0 与 axis 1 的顺序
x.norm('fro')                   # 矩阵的 Frobenius 范数
x.dist(y, p=2)                  # 矩阵 (x-y) 的 2 范数
x.matmul(y)                     # 矩阵乘法
```

需要注意，Paddle 中 Tensor 的操作符均为非 inplace 操作，即 `x.add(y)` 不会在 **tensor x** 上直接进行操作，而会返回一个新的 **Tensor** 来表示运算结果。

更多 Tensor 操作相关的 API，请参考 class paddle.Tensor

1.1.2 广播 (broadcasting)

飞桨（PaddlePaddle，以下简称 Paddle）和其他框架一样，提供的一些 API 支持广播 (broadcasting) 机制，允许在一些运算时使用不同形状的张量。通常来讲，如果有一个形状较小和一个形状较大的张量，我们希望多次使用较小的张量来对较大的张量执行一些操作，看起来像是较小形状的张量的形状首先被扩展到和较大形状的张量一致，然后做运算。值得注意的是，这期间并没有对较小形状张量的数据拷贝操作。

飞桨的广播机制主要遵循如下规则（参考 Numpy 广播机制 ）：

1. 每个张量至少为一维张量

2. 从后往前比较张量的形状，当前维度的大小要么相等，要么其中一个等于一，要么其中一个不存在

例如：

```
import paddle

x = paddle.ones((2, 3, 4))
y = paddle.ones((2, 3, 4))
# 两个张量 形状一致，可以广播
z = x + y
print(z.shape)
# [2, 3, 4]

x = paddle.ones((2, 3, 1, 5))
```

(下页继续)

(续上页)

```
y = paddle.ones((3, 4, 1))
# 从后向前依次比较：
# 第一次：y 的维度大小是 1
# 第二次：x 的维度大小是 1
# 第三次：x 和 y 的维度大小相等
# 第四次：y 的维度不存在
# 所以 x 和 y 是可以广播的
z = x + y
print(z.shape)
# [2, 3, 4, 5]

# 相反
x = paddle.ones((2, 3, 4))
y = paddle.ones((2, 3, 6))
# 此时 x 和 y 是不可广播的，因为第一次比较 4 不等于 6
# z = x + y
# InvalidArgumentError: Broadcast dimension mismatch.
```

现在我们知道什么情况下两个张量是可以广播的，两个张量进行广播语义后的结果张量的形状计算规则如下：

1. 如果两个张量的形状的长度不一致，那么需要在较小形状长度的矩阵向前添加 1，直到两个张量的形状长度相等。
2. 保证两个张量形状相等之后，每个维度上的结果维度就是当前维度上较大的那个。

例如：

```
import paddle

x = paddle.ones((2, 1, 4))
y = paddle.ones((3, 1))
z = x + y
print(z.shape)
# z 的形状：[2,3,4]

x = paddle.ones((2, 1, 4))
y = paddle.ones((3, 2))
# z = x + y
# ValueError: (InvalidArgument) Broadcast dimension mismatch.
```

1.2 升级指南

1.2.1 升级概要

飞桨 2.0 版本，相对 1.8 版本有重大升级，涉及开发方面的重要变化如下：

- 动态图功能完善，动态图模下数据表示概念为 Tensor，推荐使用动态图模式；
- API 目录体系调整，API 的命名和别名进行了统一规范化，虽然兼容老版 API，但请使用新 API 体系开发；
- 数据处理、组网方式、模型训练、多卡启动、模型保存和推理等开发流程都有了对应优化，请对应查看说明；

以上变化请仔细阅读本指南。对于已有模型的升级，我们还提供了 2.0 转换工具（见附录）提供更自动化的辅助。其他一些功能增加方面诸如动态图对量化训练、混合精度的支持、动静转换等方面不在本指南列出，具体可查看Release Note或对应文档。

1.2.2 一、动态图

推荐优先使用动态图模式

飞桨 2.0 版本将会把动态图作为默认模式（如果还想使用静态图，可通过调用 `paddle.enable_static` 切换）。

```
import paddle
```

使用 Tensor 概念表示数据

静态图模式下，由于组网时使用的数据不能实时访问，Paddle 用 Variable 来表示数据。动态图下，从直观性等角度考虑，将数据表示概念统一为 Tensor。动态图下 Tensor 的创建主要有两种方法：

1. 通过调用 paddle.to_tensor 函数，将 python scalar/list，或者 numpy.ndarray 数据转换为 Paddle 的 Tensor。具体使用方法，请查看官网的 API 文档。

```
import paddle
import numpy as np

paddle.to_tensor(1)
paddle.to_tensor((1.1, 2.2))
paddle.to_tensor(np.random.randn(3, 4))
```

1. 通过调用 `paddle.zeros`, `paddle.ones`, `paddle.full`, `paddle.arange`, `paddle.rand`, `paddle.randn`, `paddle.randint`, `paddle.normal`, `paddle.uniform` 等函数，创建并返回 Tensor。

1.2.3 二、API

API 目录结构

为了 API 组织更加简洁和清晰，将原来 padddle.fluid.xxx 的目录体系全新升级为 paddle.xxx，并对子目录的组织进行了系统的条理化优化。同时还增加了高层 API，可以高低搭配使用。paddle.fluid 目录下暂时保留了 1.8 版本 API，主要是兼容性考虑，未来会被删除。**基于 2.0 的开发任务，请使用 paddle 目录下的 API，不要再使用 paddle.fluid 目录下的 API**。如果发现 Paddle 目录下有 API 缺失的情况，推荐使用基础 API 进行组合实现；您也可以通过在 github 上提 issue 的方式向我们反馈。

2.0 版本的 API 整体目录结构如下：

API 别名规则

- 为了方便用户使用，API 会在不同的路径下建立别名：
  - 所有 device, framework, tensor 目录下的 API，均在 paddle 根目录建立别名；除少数特殊 API 外，其他 API 在 paddle 根目录下均没有别名。
  - paddle.nn 目录下除 functional 目录以外的所有 API，在 paddle.nn 目录下均有别名；functional 目录中的 API，在 paddle.nn 目录下均没有别名。
- **推荐用户优先使用较短的路径的别名**，比如 `paddle.add -> paddle.tensor.add`，推荐优先使用 `paddle.add`
- 以下为一些特殊的别名关系，推荐使用左边的 API 名称：
  - paddle.tanh -> paddle.tensor.tanh -> paddle.nn.functional.tanh
  - paddle.remainder -> paddle.mod -> paddle.floor_mod
  - paddle.rand -> paddle.uniform
  - paddle.randn -> paddle.standard_normal
  - Layer.set_state_dict -> Layer.set_dict

常用 API 名称变化

- 加、减、乘、除使用全称，不使用简称
- 对于当前逐元素操作，不加 elementwise 前缀
- 对于按照某一轴操作，不加 reduce 前缀
- Conv, Pool, Dropout, BatchNorm, Pad 组网类 API 根据输入数据类型增加 1D, 2D, 3D 后缀

1.2.4 三、开发流程

数据处理

数据处理推荐使用 **paddle.io 目录下的 Dataset，Sampler, BatchSampler, DataLoader 接口**，不推荐 reader 类接口。一些常用的数据集已经在 paddle.vision.datasets 和 paddle.text.datasets 目录实现，具体参考 API 文档。

```
from paddle.io import Dataset

class MyDataset(Dataset):
    """
    步骤一：继承 paddle.io.Dataset 类
    """
    def __init__(self, mode='train'):
        """
        步骤二：实现构造函数，定义数据读取方式，划分训练和测试数据集
        """
        super(MyDataset, self).__init__()

        if mode == 'train':
            self.data = [
                ['traindata1', 'label1'],
                ['traindata2', 'label2'],
                ['traindata3', 'label3'],
                ['traindata4', 'label4'],
            ]
        else:
            self.data = [
                ['testdata1', 'label1'],
                ['testdata2', 'label2'],
                ['testdata3', 'label3'],
                ['testdata4', 'label4'],
            ]

    def __getitem__(self, index):
        """
        步骤三：实现__getitem__ 方法，定义指定 index 时如何获取数据，并返回单条数据（训练数据，对应的标签）
        """
        data = self.data[index][0]
        label = self.data[index][1]

        return data, label
```

(下页继续)

(续上页)

```
    def __len__(self):
        """
        步骤四：实现__len__ 方法，返回数据集总数目
        """
        return len(self.data)

# 测试定义的数据集
train_dataset = MyDataset(mode='train')
val_dataset = MyDataset(mode='test')

print('=============train dataset=============')
for data, label in train_dataset:
    print(data, label)

print('=============evaluation dataset=============')
for data, label in val_dataset:
    print(data, label)
```

组网方式

Sequential 组网

针对顺序的线性网络结构我们可以直接使用 Sequential 来快速完成组网，可以减少类的定义等代码编写。

```
import paddle

# Sequential 形式组网
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)
```

SubClass 组网

针对一些比较复杂的网络结构，就可以使用 Layer 子类定义的方式来进行模型代码编写，在 `__init__` 构造函数中进行组网 Layer 的声明，在 `forward` 中使用声明的 Layer 变量进行前向计算。子类组网方式也可以实现 sublayer 的复用，针对相同的 layer 可以在构造函数中一次性定义，在 forward 中多次调用。

```
import paddle

# Layer 类继承方式组网
class Mnist(paddle.nn.Layer):
    def __init__(self):
        super(Mnist, self).__init__()

        self.flatten = paddle.nn.Flatten()
        self.linear_1 = paddle.nn.Linear(784, 512)
        self.linear_2 = paddle.nn.Linear(512, 10)
        self.relu = paddle.nn.ReLU()
        self.dropout = paddle.nn.Dropout(0.2)

    def forward(self, inputs):
        y = self.flatten(inputs)
        y = self.linear_1(y)
        y = self.relu(y)
        y = self.dropout(y)
        y = self.linear_2(y)

        return y

mnist = Mnist()
```

模型训练

使用高层 API

增加了 paddle.Model 高层 API，大部分任务可以使用此 API 用于简化训练、评估、预测类代码开发。注意区别 Model 和 Net 概念，Net 是指继承 paddle.nn.Layer 的网络结构；而 Model 是指持有一个 Net 对象，同时指定损失函数、优化算法、评估指标的可训练、评估、预测的实例。具体参考高层 API 的代码示例。

```
import paddle
from paddle.vision.transforms import ToTensor

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
```

(下页继续)

(续上页)

```
lenet = paddle.vision.models.LeNet()

# Mnist 继承 paddle.nn.Layer 属于 Net, model 包含了训练功能
model = paddle.Model(lenet)

# 设置训练模型所需的 optimizer, loss, metric
model.prepare(
    paddle.optimizer.Adam(learning_rate=0.001, parameters=model.parameters()),
    paddle.nn.CrossEntropyLoss(),
    paddle.metric.Accuracy(topk=(1, 2))
    )

# 启动训练
model.fit(train_dataset, epochs=2, batch_size=64, log_freq=200)

# 启动评估
model.evaluate(test_dataset, log_freq=20, batch_size=64)
```

使用基础 API

```
import paddle
from paddle.vision.transforms import ToTensor

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
lenet = paddle.vision.models.LeNet()
loss_fn = paddle.nn.CrossEntropyLoss()

# 加载训练集 batch_size 设为 64
train_loader = paddle.io.DataLoader(train_dataset, batch_size=64, shuffle=True)

def train():
    epochs = 2
    adam = paddle.optimizer.Adam(learning_rate=0.001, parameters=lenet.parameters())
    # 用 Adam 作为优化函数
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = data[1]
            predicts = lenet(x_data)
            acc = paddle.metric.accuracy(predicts, y_data)
            loss = loss_fn(predicts, y_data)
```

(下页继续)

(续上页)

```
            loss.backward()
            if batch_id % 100 == 0:
                print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,
↪ batch_id, loss.numpy(), acc.numpy()))
            adam.step()
            adam.clear_grad()

# 启动训练
train()
```

单机多卡启动

2.0 增加 paddle.distributed.spawn 函数来启动单机多卡训练，同时原有的 paddle.distributed.launch 的方式依然保留。

方式 1、launch 启动

高层 API 场景

当调用 paddle.Model 高层来实现训练时，想要启动单机多卡训练非常简单，代码不需要做任何修改，只需要在启动时增加一下参数`-m paddle.distributed.launch`。

```
# 单机单卡启动，默认使用第 0 号卡
$ python train.py

# 单机多卡启动，默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py

# 单机多卡启动，设置当前使用的第 0 号和第 1 号卡
$ python -m paddle.distributed.launch --selected_gpus='0,1' train.py

# 单机多卡启动，设置当前使用第 0 号和第 1 号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

基础 API 场景

如果使用基础 API 实现训练，想要启动单机多卡训练，需要对单机单卡的代码进行 3 处修改，具体如下：

```
import paddle
# 第 1 处改动，导入分布式训练所需要的包
import paddle.distributed as dist

train_dataset = paddle.vision.datasets.MNIST(mode='train')
test_dataset = paddle.vision.datasets.MNIST(mode='test')
lenet = paddle.vision.models.LeNet()
loss_fn = paddle.nn.CrossEntropyLoss()

# 加载训练集 batch_size 设为 64
train_loader = paddle.io.DataLoader(train_dataset, batch_size=64, shuffle=True)

def train():
    # 第 2 处改动，初始化并行环境
    dist.init_parallel_env()

    # 第 3 处改动，增加 paddle.DataParallel 封装
    lenet = paddle.DataParallel(lenet)
    epochs = 2
    adam = paddle.optimizer.Adam(learning_rate=0.001, parameters=lenet.parameters())
    # 用 Adam 作为优化函数
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = data[1]
            predicts = lenet(x_data)
            acc = paddle.metric.accuracy(predicts, y_data)
            loss = loss_fn(predicts, y_data)
            loss.backward()
            if batch_id % 100 == 0:
                print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,
↪ batch_id, loss.numpy(), acc.numpy()))
            adam.step()
            adam.clear_grad()

# 启动训练
train()
```

修改完后保存文件，然后使用跟高层 API 相同的启动方式即可

注意：单卡训练不支持调用 `init_parallel_env`，请使用以下几种方式进行分布式训练。

```
# 单机多卡启动，默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py

# 单机多卡启动，设置当前使用的第 0 号和第 1 号卡
$ python -m paddle.distributed.launch --selected_gpus '0,1' train.py

# 单机多卡启动，设置当前使用第 0 号和第 1 号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

方式 2、spawn 启动

launch 方式启动训练，以文件为单位启动多进程，需要用户在启动时调用 `paddle.distributed.launch`，对于进程的管理要求较高。飞桨框架 2.0 版本增加了 spawn 启动方式，可以更好地控制进程，在日志打印、训练退出时更友好。使用示例如下：

```
from __future__ import print_function

import paddle
import paddle.nn as nn
import paddle.optimizer as opt
import paddle.distributed as dist

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear1 = nn.Linear(10, 10)
        self._linear2 = nn.Linear(10, 1)

    def forward(self, x):
        return self._linear2(self._linear1(x))

def train(print_result=False):

    # 1. 初始化并行训练环境
    dist.init_parallel_env()

    # 2. 创建并行训练 Layer 和 Optimizer
    layer = LinearNet()
    dp_layer = paddle.DataParallel(layer)

    loss_fn = nn.MSELoss()
    adam = opt.Adam(
```

(下页继续)

(续上页)

```
        learning_rate=0.001, parameters=dp_layer.parameters())

    # 3. 运行网络
    inputs = paddle.randn([10, 10], 'float32')
    outputs = dp_layer(inputs)
    labels = paddle.randn([10, 1], 'float32')
    loss = loss_fn(outputs, labels)

    if print_result is True:
        print("loss:", loss.numpy())

    loss.backward()

    adam.step()
    adam.clear_grad()

# 使用方式 1: 仅传入训练函数
# 适用场景: 训练函数不需要任何参数，并且需要使用所有当前可见的 GPU 设备并行训练
if __name__ == '__main__':
    dist.spawn(train)

# 使用方式 2: 传入训练函数和参数
# 适用场景: 训练函数需要一些参数，并且需要使用所有当前可见的 GPU 设备并行训练
if __name__ == '__main__':
    dist.spawn(train, args=(True,))

# 使用方式 3: 传入训练函数、参数并指定并行进程数
# 适用场景: 训练函数需要一些参数，并且仅需要使用部分可见的 GPU 设备并行训练，例如:
# 当前机器有 8 张 GPU 卡 {0,1,2,3,4,5,6,7}，此时会使用前两张卡 {0,1};
# 或者当前机器通过配置环境变量 CUDA_VISIBLE_DEVICES=4,5,6,7，仅使 4 张
# GPU 卡可见，此时会使用可见的前两张卡 {4,5}
if __name__ == '__main__':
    dist.spawn(train, args=(True,), nprocs=2)

# 使用方式 4: 传入训练函数、参数、指定进程数并指定当前使用的卡号
# 使用场景: 训练函数需要一些参数，并且仅需要使用部分可见的 GPU 设备并行训练，但是
# 可能由于权限问题，无权配置当前机器的环境变量，例如: 当前机器有 8 张 GPU 卡
# {0,1,2,3,4,5,6,7}，但你无权配置 CUDA_VISIBLE_DEVICES，此时可以通过
# 指定参数 selected_gpus 选择希望使用的卡，例如 selected_gpus='4,5',
# 可以指定使用第 4 号卡和第 5 号卡
if __name__ == '__main__':
    dist.spawn(train, nprocs=2, selected_gpus='4,5')
```

(下页继续)

(续上页)

```
# 使用方式 5: 指定多卡通信的起始端口
# 使用场景: 端口建立通信时提示需要重试或者通信建立失败
# Paddle 默认会通过在当前机器上寻找空闲的端口用于多卡通信，但当机器使用环境
# 较为复杂时，程序找到的端口可能不够稳定，此时可以自行指定稳定的空闲起始
# 端口以获得更稳定的训练体验
if __name__ == '__main__':
    dist.spawn(train, nprocs=2, started_port=12345)
```

模型保存

Paddle 保存的模型有两种格式，一种是训练格式，保存模型参数和优化器相关的状态，可用于恢复训练；一种是预测格式，保存预测的静态图网络结构以及参数，用于预测部署。

高层 API 场景

高层 API 下用于预测部署的模型保存方法为：

```
model = paddle.Model(Mnist())
# 预测格式，保存的模型可用于预测部署
model.save('mnist', training=False)
# 保存后可以得到预测部署所需要的模型
```

基础 API 场景

动态图训练的模型，可以通过动静转换功能，转换为可部署的静态图模型，具体做法如下：

```
import paddle
from paddle.jit import to_static
from paddle.static import InputSpec

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    # 第 1 处改动
    # 通过 InputSpec 指定输入数据的形状，None 表示可变长
    # 通过 to_static 装饰器将动态图转换为静态图 Program
    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), InputSpec(shape=[3],
↪ name='y')])
    def forward(self, x, y):
```

(下页继续)

(续上页)

```
        out = self.linear(x)
        out = out + y
        return out


net = SimpleNet()

# 第 2 处改动
# 保存静态图模型，可用于预测部署
paddle.jit.save(net, './simple_net')
```

推理

推理库 Paddle Inference 的 API 做了升级，简化了写法，以及去掉了历史上冗余的概念。API 的变化为纯增，原有 API 保持不变，但推荐新的 API 体系，旧 API 在后续版本会逐步删除。

C++ API

重要变化：

- 命名空间从 `paddle` 变更为 `paddle_infer`
- `PaddleTensor`, `PaddleBuf` 等被废弃，`ZeroCopyTensor` 变为默认 Tensor 类型，并更名为 `Tensor`
- 新增 `PredictorPool` 工具类简化多线程 predictor 的创建，后续也会增加更多周边工具
- `CreatePredictor` (原 `CreatePaddlePredictor`) 的返回值由 `unique_ptr` 变为 `shared_ptr` 以避免 Clone 后析构顺序出错的问题

API 变更

使用新 C++ API 的流程与之前完全一致，只有命名变化

```
#include "paddle_infernce_api.h"
using namespace paddle_infer;

Config config;
config.SetModel("xxx_model_dir");

auto predictor = CreatePredictor(config);

// Get the handles for the inputs and outputs of the model
auto input0 = predictor->GetInputHandle("X");
auto output0 = predictor->GetOutputHandle("Out");
```

(下页继续)

(续上页)

```
for (...) {
  // Assign data to input0
  MyServiceSetData(input0);

  predictor->Run();

  // get data from the output0 handle
  MyServiceGetData(output0);
}
```

Python API

Python API 的变更与 C++ 基本对应，会在 2.0 版发布。

1.2.5 附录

2.0 转换工具

为了降级代码升级的成本，我们提供了转换工具，可以帮助将 Paddle 1.8 版本开发的代码，升级为 2.0 的 API。由于相比于 Paddle 1.8 版本，2.0 版本的 API 进行了大量的升级，包括 API 名称，参数名称，行为等。转换工具当前还不能覆盖所有的 API 升级；对于无法转换的 API，转换工具会报错，提示用户手动升级。

https://github.com/PaddlePaddle/paddle_upgrade_tool

对于转换工具没有覆盖的 API，请查看官网的 API 文档，手动升级代码的 API。

2.0 文档教程

以下提供了 2.0 版本的一些示例教程：

您可以在官网应用实践栏目内进行在线浏览，也可以下载在这里提供的源代码：https://github.com/PaddlePaddle/book/tree/develop/paddle2.0_docs

1.3 版本迁移工具

在飞桨框架 2.0 中，我们 API 的位置、命名、参数、行为，进行了系统性的调整和规范，将 API 体系从 1.X 版本的 `paddle.fluid.*` 迁移到了 `paddle.*` 下。paddle.fluid 目录下暂时保留了 1.8 版本 API，主要是兼容性考虑，未来会被删除。

1.3.1 使用版本迁移工具自动迁移您的 Paddle 1.x 的代码到 Paddle 2.0 的代码

WARNING: 版本自动迁移工具并不能处理所有的情况，在使用本工具后，您仍然需要手工来进行检查并做相应的调整。

安装

版本迁移工具可以通过 pip 的方式安装，方式如下:

```
$ pip install paddle_upgrade_tool
```

基本用法

paddle_upgrade_tool 可以使用下面的方式，快速使用:

```
$ paddle_upgrade_tool --inpath /path/to/model.py
```

这将在命令行中，以 `diff` 的形式，展示 model.py 从 Paddle 1.x 转换为 Paddle 2.0 的变化。如果您确认上述变化没有问题，只需要再执行:

```
$ paddle_upgrade_tool --inpath /path/to/model.py --write
```

就会原地改写 model.py，将上述变化改写到您的源文件中。注意: 我们会默认备份源文件，到 ~/.paddle_upgrade_tool/下。

参数说明如下:

- –inpath 输入文件路径，可以为单个文件或文件夹。
- –write 是否原地修改输入的文件，默认值 False，表示不修改。如果为 True，表示对文件进行原地修改。添加此参数也表示对文件进行原地修改。
- –backup 可选，是否备份源文件，默认值为 `~/.paddle_upgrade_tool/`，在此路径下备份源文件。
- –no-log-file 可选，是否需要输出日志文件，默认值为 False，即输出日志文件。
- –log-filepath 可选，输出日志的路径，默认值为 `report.log`，输出日志文件的路径。
- –no-confirm 可选，输入文件夹时，是否逐文件确认原地写入，只在`--write` 为 True 时有效，默认值为 False，表示需要逐文件确认。
- –parallel 可选，控制转换文件的并发数，当 `no-confirm` 为 True 时不生效，默认值:`None`。
- –log-level 可选，log 级别，可为 [‘DEBUG’，‘INFO’，‘WARNING’，‘ERROR’] 默认值：`INFO`。
- –refactor 可选，debug 时使用。
- –print-match 可选，debug 时使用。

使用教程

开始

在使用 paddle_upgrade_tool 前，需要确保您已经安装了 Paddle 2.0 版本。

```
import paddle
print (paddle.__version__)
```

```
2.0.0
```

克隆paddlePaddle/models来作为工具的测试。

```
$ git clone https://github.com/PaddlePaddle/models
```

```
Cloning into 'models'...
remote: Enumerating objects: 8, done.[K
remote: Counting objects: 100% (8/8), done.[K
remote: Compressing objects: 100% (8/8), done.[K
remote: Total 35011 (delta 1), reused 0 (delta 0), pack-reused 35003[K
Receiving objects: 100% (35011/35011), 356.97 MiB | 1.53 MiB/s, done.
Resolving deltas: 100% (23291/23291), done.
```

查看帮助文档

您可以直接通过下面的方式，查看帮助文档。

```
$ paddle_upgrade_tool -h
```

```
usage: paddle_upgrade_tool [-h] [--log-level {DEBUG,INFO,WARNING,ERROR}]
                           [--no-log-file] [--log-filepath LOG_FILEPATH] -i
                           INPATH [-b [BACKUP]] [-w] [--no-confirm]
                           [-p PARALLEL]
                           [-r {refactor_import,norm_api_alias,args_to_kwargs,
↪refactor_kwargs,api_rename,refactor_with,post_refactor}]
                           [--print-match]

optional arguments:
  -h, --help            show this help message and exit
  --log-level {DEBUG,INFO,WARNING,ERROR}
                        set log level, default is INFO
  --no-log-file         don't log to file
  --log-filepath LOG_FILEPATH
```

(下页继续)

(续上页)

```
                        set log file path, default is "report.log"
  -i INPATH, --inpath INPATH
                        the file or directory path you want to upgrade.
  -b [BACKUP], --backup [BACKUP]
                        backup directory, default is the
                        "~/.paddle_upgrade_tool/".
  -w, --write           modify files in-place.
  --no-confirm          write files in-place without confirm, ignored without
                        --write.
  -p PARALLEL, --parallel PARALLEL
                        specify the maximum number of concurrent processes to
                        use when refactoring, ignored with --no-confirm.
  -r {refactor_import,norm_api_alias,args_to_kwargs,refactor_kwargs,api_rename,
↪refactor_with,post_refactor}, --refactor {refactor_import,norm_api_alias,args_to_
↪kwargs,refactor_kwargs,api_rename,refactor_with,post_refactor}
                        this is a debug option. Specify refactor you want to
                        run. If none, all refactors will be run.
  --print-match         this is a debug option. Print matched code and node
                        for each file.
```

Paddle 1.x 的例子

这里是一个基于 Paddle 1.x 实现的一个 mnist 分类，部分内容如下：

```
$ head -n 198 models/dygraph/mnist/train.py | tail -n  20
```

```
with fluid.dygraph.guard(place):
    if args.ce:
        print("ce mode")
        seed = 33
        np.random.seed(seed)
        fluid.default_startup_program().random_seed = seed
        fluid.default_main_program().random_seed = seed

    if args.use_data_parallel:
        strategy = fluid.dygraph.parallel.prepare_context()
    mnist = MNIST()
    adam = AdamOptimizer(learning_rate=0.001, parameter_list=mnist.parameters())
    if args.use_data_parallel:
        mnist = fluid.dygraph.parallel.DataParallel(mnist, strategy)

    train_reader = paddle.batch(
```

(下页继续)

(续上页)

```
        paddle.dataset.mnist.train(), batch_size=BATCH_SIZE, drop_last=True)
    if args.use_data_parallel:
        train_reader = fluid.contrib.reader.distributed_batch_reader(
            train_reader)
```

使用 paddle_upgrade_tool 进行转化

paddle_upgrade_tool 支持单文件的转化，您可以通过下方的命令直接转化单独的文件。

```
$ paddle_upgrade_tool --inpath models/dygraph/mnist/train.py
```

注意，对于参数的删除及一些特殊情况，我们都会打印 WARNING 信息，需要您仔细核对相关内容。如果您觉得上述信息没有问题，可以直接对文件进行原地修改，方式如下：

```
$ paddle_upgrade_tool --inpath models/dygraph/mnist/train.py --write
```

此时，命令行会弹出下方的提示：

```
"models/dygraph/mnist/train.py" will be modified in-place, and it has been backed up␣
↪to "~/.paddle_upgrade_tool/train.py_backup_2020_09_09_20_35_15_037821". Do you want␣
↪to continue? [Y/n]:
```

输入 y 后即开始执行代码迁移。为了高效完成迁移，我们这里采用了原地写入的方式。此外，为了防止特殊情况，我们会备份转换前的代码到 ~/.paddle_upgrade_tool 目录下，如果需要，您可以在备份目录下找到转换前的代码。

代码迁移完成后，会生成一个 report.log 文件，记录了迁移的详情。内容如下：

```
$ cat report.log
```

注意事项

- 本迁移工具不能完成所有 API 的迁移，有少量的 API 需要您手动完成迁移，具体信息可见 WARNING。

使用 Paddle 2.0

完成迁移后，代码就从 Paddle 1.x 迁移到了 Paddle 2.0，您就可以在 Paddle 2.0 下进行相关的开发。

CHAPTER 2

模型开发

本部分将介绍飞桨框架 2.0 的开发流程。

为了快速上手飞桨框架 2.0，你可以参考 10 分钟快速上手飞桨；

当完成了快速上手的任务后，下面这些模块会阐述如何用飞桨框架 2.0，实现深度学习过程中的每一步。具体包括：

- 数据集定义与加载：飞桨框架数据加载的方式，主要为 `paddle.io.Dataset` + `paddle.io.DataLoader`，以及飞桨内置数据集的介绍。
- 数据预处理：飞桨框架数据预处理的方法，主要是 `paddle.vision.transform.*`。
- 模型组网：飞桨框架组网 API 的介绍，主要是 `paddle.nn.*`，然后是飞桨框架组网方式的介绍，即 Sequential 的组网与 SubClass 的组网。
- 训练与预测：飞桨框架训练与预测的方法，有两种方式，一种是使用高层 API`paddle.Model` 封装模型，然后调用 `model.fit()`、`model.evaluate()`、`model.predict()` 完成模型的训练与预测；另一种是用基础 API 完成模型的训练与预测，也就是对高层 API 的拆解。
- 资源配置：飞桨框架在单机单卡、单机多卡的场景下完成模型的训练与预测。
- 自定义指标：飞桨框架自定义指标的方法，主要包含自定义 Loss、自定义 Metric 与自定义 Callback。
- 模型的加载与保存：飞桨框架模型的加载与保存体系介绍。
- 模型转 ONNX 协议：飞桨框架模型转换为 ONNX 格式介绍。

2.1 10 分钟快速上手飞桨（PaddlePaddle）

本示例通过一个基础案例，带你快速了解如何使用飞桨框架。

2.1.1 一、安装飞桨

如果你已经安装好飞桨那么可以跳过此步骤。飞桨提供了一个方便易用的安装引导页面，你可以通过选择自己的系统和软件版本来获取对应的安装命令，具体可以点击快速安装查看。

2.1.2 二、导入飞桨

安装好飞桨后你就可以在 Python 程序中导入飞桨。

```
import paddle
print(paddle.__version__)
```

```
2.0.0
```

2.1.3 三、实践：手写数字识别任务

简单的说，深度学习任务一般分为几个核心步骤：1. 数据集的准备和加载；2. 模型构建；3. 模型训练；4. 模型评估。接下来你可以使用飞桨框架 API，一步步实现上述步骤。

3.1 加载内置数据集

飞桨框架内置了一些常见的数据集，在这个示例中，你可以加载飞桨框架的内置数据集：手写数字体数据集。这里加载两个数据集，一个用来训练模型，一个用来评估模型。

```
from paddle.vision.transforms import ToTensor

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
val_dataset =  paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
```

3.2 模型搭建

通过 Sequential 将一层一层的网络结构组建起来。注意，需要先对数据进行 Flatten 操作，将 [1, 28, 28] 形状的图片数据改变形状为 [1, 784]。

```
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)
```

3.3 模型训练

在训练模型前，需要配置训练模型时损失的计算方法与优化方法，你可以使用飞桨框架提供的 prepare 完成，之后使用 fit 接口来开始训练模型。

```
# 预计模型结构生成模型对象，便于进行后续的配置、训练和验证
model = paddle.Model(mnist)

# 模型训练相关配置，准备损失计算方法，优化器和精度计算方法
model.prepare(paddle.optimizer.Adam(parameters=model.parameters()),
              paddle.nn.CrossEntropyLoss(),
              paddle.metric.Accuracy())

# 开始模型训练
model.fit(train_dataset,
          epochs=5,
          batch_size=64,
          verbose=1)
```

```
The loss value printed in the log is the current step, and the metric is the average␣
↪value of previous step.
Epoch 1/5
step 938/938 [==============================] - loss: 0.1358 - acc: 0.9284 - 18ms/step
Epoch 2/5
step 938/938 [==============================] - loss: 0.0370 - acc: 0.9680 - 18ms/step
Epoch 3/5
step 938/938 [==============================] - loss: 0.0284 - acc: 0.9780 - 18ms/step
Epoch 4/5
step 938/938 [==============================] - loss: 0.0062 - acc: 0.9823 - 18ms/step
Epoch 5/5
step 938/938 [==============================] - loss: 0.0924 - acc: 0.9859 - 18ms/step
```

3.4 模型评估

你可以使用预先定义的验证数据集来评估前一步训练得到的模型的精度。

```
model.evaluate(val_dataset, verbose=0)
```

```
{'loss': [0.0], 'acc': 0.9804}
```

可以看出，初步训练得到的模型效果在 98% 附近，在逐渐了解飞桨后，你可以通过调整其中的训练参数来提升模型的精度。

至此你就通过飞桨几个简单的 API 完成了一个深度学习任务，你也可以针对自己的需求来更换其中的代码，比如对数据集进行增强、使用 CNN 模型等，飞桨官网提供了丰富的教程与案例可供参考。

2.2 数据集定义与加载

深度学习模型在训练时需要大量的数据来完成模型调优，这个过程均是数字的计算，无法直接使用原始图片和文本等来完成计算。因此与需要对原始的各种数据文件进行处理，转换成深度学习模型可以使用的数据类型。

2.2.1 一、框架自带数据集

飞桨框架将深度学习任务中常用到的数据集作为领域 API 开放，对应 API 所在目录为 `paddle.vision.datasets` 与 `paddle.text.datasets`，你可以通过以下代码飞桨框架中提供了哪些数据集。

```
import paddle
print(' 视觉相关数据集：', paddle.vision.datasets.__all__)
print(' 自然语言相关数据集：', paddle.text.datasets.__all__)
```

```
视觉相关数据集： ['DatasetFolder', 'ImageFolder', 'MNIST', 'FashionMNIST', 'Flowers',
↪'Cifar10', 'Cifar100', 'VOC2012']
自然语言相关数据集： ['Conll05st', 'Imdb', 'Imikolov', 'Movielens', 'UCIHousing', 'WMT14',
↪ 'WMT16']
```

警告： 除 `paddle.vision.dataset` 与 `paddle.text.dataset` 外，飞桨框架还内置了另一套数据集，路径为 `paddle.dataset.*`，但是该数据集的使用方式较老，会在未来的版本废弃，请尽量不要使用该目录下数据集的 API。

这里你可以定义手写数字体的数据集，其他数据集的使用方式也都类似。用 `mode` 来标识训练集与测试集。数据集接口会自动从远端下载数据集到本机缓存目录 `~/.cache/paddle/dataset`。

```
from paddle.vision.transforms import ToTensor
# 训练数据集 用 ToTensor 将数据格式转为 Tensor
train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())

# 验证数据集
val_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
```

2.2.2 二、自定义数据集

在实际的场景中，更多需要使用你已有的相关数据来定义数据集。你可以使用飞桨提供的 paddle.io.Dataset 基类，来快速实现自定义数据集。

```
import paddle
from paddle.io import Dataset

BATCH_SIZE = 64
BATCH_NUM = 20

IMAGE_SIZE = (28, 28)
CLASS_NUM = 10


class MyDataset(Dataset):
    """
    步骤一：继承 paddle.io.Dataset 类
    """
    def __init__(self, num_samples):
        """
        步骤二：实现构造函数，定义数据集大小
        """
        super(MyDataset, self).__init__()
        self.num_samples = num_samples

    def __getitem__(self, index):
        """
        步骤三：实现__getitem__ 方法，定义指定 index 时如何获取数据，并返回单条数据（训练数据，对应
的标签）
        """
        data = paddle.uniform(IMAGE_SIZE, dtype='float32')
        label = paddle.randint(0, CLASS_NUM-1, dtype='int64')

        return data, label
```

(下页继续)

(续上页)

```
    def __len__(self):
        """
        步骤四：实现__len__ 方法，返回数据集总数目
        """
        return self.num_samples

# 测试定义的数据集
custom_dataset = MyDataset(BATCH_SIZE * BATCH_NUM)

print('=============custom dataset=============')
for data, label in custom_dataset:
    print(data.shape, label.shape)
    break
```

```
=============custom dataset=============
[28, 28] [1]
```

通过以上的方式，你就可以根据实际场景，构造自己的数据集。

2.2.3 三、数据加载

飞桨推荐使用 `paddle.io.DataLoader` 完成数据的加载。简单的示例如下：

```
train_loader = paddle.io.DataLoader(custom_dataset, batch_size=BATCH_SIZE,␣
↪shuffle=True)
# 如果要加载内置数据集，将 custom_dataset 换为 train_dataset 即可
for batch_id, data in enumerate(train_loader()):
    x_data = data[0]
    y_data = data[1]

    print(x_data.shape)
    print(y_data.shape)
    break
```

```
[64, 28, 28]
[64, 1]
```

通过上述的方法，你就定义了一个数据迭代器 `train_loader`，用于加载训练数据。通过 `batch_size=64` 设置了数据集的批大小为 64，通过 `shuffle=True`，在取数据前会打乱数据。此外，你还可以通过设置 `num_workers` 来开启多进程数据加载，提升加载速度。

注解： DataLoader 默认用异步加载数据的方式来读取数据，一方面可以提升数据加载的速度，另一方面也会

占据更少的内存。如果你需要同时加载全部数据到内存中，请设置 `use_buffer_reader=False`。

2.3 数据预处理

训练过程中有时会遇到过拟合的问题，其中一个解决方法就是对训练数据做增强，对数据进行处理得到不同的图像，从而泛化数据集。数据增强 API 是定义在领域目录的 transofrms 下，这里介绍两种使用方式，一种是基于框架内置数据集，一种是基于自定义的数据集。

2.3.1 一、飞桨框架内置数据集

针对飞桨框架内置图像数据集的预处理，飞桨框架将这部分 API 整合到 `paddle.vision.transforms` 下，你可以通过以下方式查看：

```
import paddle
print(' 数据处理方法：', paddle.vision.transforms.__all__)
```

```
数据处理方法： ['BaseTransform', 'Compose', 'Resize', 'RandomResizedCrop', 'CenterCrop',
↪'RandomHorizontalFlip', 'RandomVerticalFlip', 'Transpose', 'Normalize',
↪'BrightnessTransform', 'SaturationTransform', 'ContrastTransform', 'HueTransform',
↪'ColorJitter', 'RandomCrop', 'Pad', 'RandomRotation', 'Grayscale', 'ToTensor', 'to_
↪tensor', 'hflip', 'vflip', 'resize', 'pad', 'rotate', 'to_grayscale', 'crop',
↪'center_crop', 'adjust_brightness', 'adjust_contrast', 'adjust_hue', 'normalize']
```

你可以同构以下方式随机调整图像的亮度、对比度、饱和度，并调整图像的大小，对图像的其他调整，可以参考相关的 API 文档。

```
from paddle.vision.transforms import Compose, Resize, ColorJitter

# 定义想要使用的数据增强方式，这里包括随机调整亮度、对比度和饱和度，改变图片大小
transform = Compose([ColorJitter(), Resize(size=32)])

# 通过 transform 参数传递定义好的数据增强方法即可完成对自带数据集的增强
train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=transform)
```

2.3.2 二、自定义数据集

对于自定义的数据集，你可以在数据集的构造函数中进行数据增强方法的定义，之后对 __getitem__ 中返回的数据进行应用，就可以完成自定义数据增强。

```
import paddle
from paddle.io import Dataset
from paddle.vision.transforms import Compose, Resize

BATCH_SIZE = 64
BATCH_NUM = 20

IMAGE_SIZE = (28, 28)
CLASS_NUM = 10

class MyDataset(Dataset):
    def __init__(self, num_samples):
        super(MyDataset, self).__init__()
        self.num_samples = num_samples
        # 在 `__init__` 中定义数据增强方法，此处为调整图像大小
        self.transform = Compose([Resize(size=32)])

    def __getitem__(self, index):
        data = paddle.uniform(IMAGE_SIZE, dtype='float32')
        # 在 `__getitem__` 中对数据集使用数据增强方法
        data = self.transform(data.numpy())

        label = paddle.randint(0, CLASS_NUM-1, dtype='int64')

        return data, label

    def __len__(self):
        return self.num_samples

# 测试定义的数据集
custom_dataset = MyDataset(BATCH_SIZE * BATCH_NUM)

print('=============custom dataset=============')
for data, label in custom_dataset:
    print(data.shape, label.shape)
    break
```

```
=============custom dataset=============
[32, 32] [1]
```

可以看出，输出的形状从 [28, 28, 1] 变为了 [32, 32, 1]，证明完成了图像的大小调整。

2.4 模型组网

完成数据集的构建后，需要构建网络模型。首先介绍飞桨组网相关的 API，主要是 paddle.nn 下的 API 介绍，然后介绍动态图下飞桨框架支持的两种组网方式，分别为 Sequential 组网与 SubClass 组网，最后，介绍飞桨框架内置的算法模型。

2.4.1 一、paddle.nn 简介

飞桨框架 2.0 中，组网相关的 API 都在 paddle.nn 目录下，你可以通过 Sequential 或 SubClass 的方式构建具体的模型。组网相关的 API 类别与具体的 API 列表如下表：

功能	API 名称
Conv	Conv1D、Conv2D、Conv3D、Conv1DTranspose、Conv2DTranspose、Conv3DTranspose
Pool	AdaptiveAvgPool1D、AdaptiveAvgPool2D、AdaptiveAvgPool3D、AdaptiveMaxPool1D、AdaptiveMaxPool2D、AdaptiveMaxPool3D、AvgPool1D、AvgPool2D、AvgPool3D、MaxPool1D、MaxPool2D、MaxPool3D
Padding	Pad1D、Pad2D、Pad3d
Activation	ELU、GELU、Hardshrink、Hardtanh、HSigmoid、LeakyReLU、LogSigmoid、LogSoftmax、PReLU、ReLU、ReLU6、SELU、Sigmoid、Softmax、Softplus、Softshrink、Softsign、Tanh、Tanhshrink
Normlization	BatchNorm、BatchNorm1D、BatchNorm2D、BatchNorm3D、GroupNorm、InstanceNorm1D、InstanceNorm2D、InstanceNorm3D、LayerNorm、SpectralNorm、SyncBatchNorm
Recurrent NN	BiRNN、GRU、GRUCell、LSTM、LSTMCell、RNN、RNNCellBase、SimpleRNN、SimpleRNNCell
Transformer	Transformer、TransformerDecoder、TransformerDecoderLayer、TransformerEncoder、TransformerEncoderLayer
Dropout	AlphaDropout、Dropout、Dropout2d、Dropout3d
Loss	BCELoss、BCEWithLogitsLoss、CrossEntropyLoss、CTCLoss、KLDivLoss、L1Loss MarginRankingLoss、MSELoss、NLLLoss、SmoothL1Loss

2.4.2 二、Sequential 组网

针对顺序的线性网络结构你可以直接使用 Sequential 来快速完成组网，可以减少类的定义等代码编写。具体代码如下：

```
import paddle
# Sequential 形式组网
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)
```

2.4.3 三、SubClass 组网

针对一些比较复杂的网络结构，就可以使用 Layer 子类定义的方式来进行模型代码编写，在 `__init__` 构造函数中进行组网 Layer 的声明，在 `forward` 中使用声明的 Layer 变量进行前向计算。子类组网方式也可以实现 sublayer 的复用，针对相同的 layer 可以在构造函数中一次性定义，在 forward 中多次调用。

```
# Layer 类继承方式组网
class Mnist(paddle.nn.Layer):
    def __init__(self):
        super(Mnist, self).__init__()

        self.flatten = paddle.nn.Flatten()
        self.linear_1 = paddle.nn.Linear(784, 512)
        self.linear_2 = paddle.nn.Linear(512, 10)
        self.relu = paddle.nn.ReLU()
        self.dropout = paddle.nn.Dropout(0.2)

    def forward(self, inputs):
        y = self.flatten(inputs)
        y = self.linear_1(y)
        y = self.relu(y)
        y = self.dropout(y)
        y = self.linear_2(y)

        return y

mnist_2 = Mnist()
```

2.4.4 四、飞桨框架内置模型

你除了可以通过上述方式组建模型外，还可以使用飞桨框架内置的模型，路径为 paddle.vision.models，具体列表如下：

```
print(' 飞桨框架内置模型: ', paddle.vision.models.__all__)
```

```
飞桨框架内置模型:  ['ResNet', 'resnet18', 'resnet34', 'resnet50', 'resnet101', 'resnet152
↪', 'VGG', 'vgg11', 'vgg13', 'vgg16', 'vgg19', 'MobileNetV1', 'mobilenet_v1',
↪'MobileNetV2', 'mobilenet_v2', 'LeNet']
```

使用方式如下：

```
lenet = paddle.vision.models.LeNet()
```

你可以通过 paddle.summary() 方法查看模型的结构与每一层输入输出形状，具体如下：

```
paddle.summary(lenet, (64, 1, 28, 28))
```

```
---------------------------------------------------------------------------
 Layer (type)       Input Shape          Output Shape         Param #
===========================================================================
   Conv2D-1      [[64, 1, 28, 28]]     [64, 6, 28, 28]           60
    ReLU-1       [[64, 6, 28, 28]]     [64, 6, 28, 28]            0
  MaxPool2D-1    [[64, 6, 28, 28]]     [64, 6, 14, 14]            0
   Conv2D-2      [[64, 6, 14, 14]]     [64, 16, 10, 10]        2,416
    ReLU-2       [[64, 16, 10, 10]]    [64, 16, 10, 10]           0
  MaxPool2D-2    [[64, 16, 10, 10]]     [64, 16, 5, 5]            0
   Linear-1         [[64, 400]]            [64, 120]           48,120
   Linear-2         [[64, 120]]             [64, 84]           10,164
   Linear-3          [[64, 84]]             [64, 10]             850
===========================================================================
Total params: 61,610
Trainable params: 61,610
Non-trainable params: 0
---------------------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 7.03
Params size (MB): 0.24
Estimated Total Size (MB): 7.46
---------------------------------------------------------------------------
```

2.5 训练与预测

在完成数据预处理，数据加载与模型的组建后，你就可以进行模型的训练与预测了。飞桨框架提供了两种训练与预测的方法，一种是用 `paddle.Model` 对模型进行封装，通过高层 API 如 `Model.fit()`、`Model.evaluate()`、`Model.predict()` 等完成模型的训练与预测；另一种就是基于基础 API 常规的训练方式。

注解： 高层 API 实现的模型训练与预测如 `Model.fit()`、`Model.evaluate()`、`Model.predict()` 都可以通过基础 API 实现，本文先介绍高层 API 的训练方式，然后会将高层 API 拆解为基础 API 的方式，方便对比学习。

2.5.1 一、训练前准备

在封装模型前，需要先完成数据的加载与模型的组建，由于这一部分高层 API 与基础 API 通用，所以都可用下面的代码实现：

```
import paddle
from paddle.vision.transforms import ToTensor

# 加载数据集
train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())

# 定义网络结构
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(1, -1),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)
```

通过上述的代码，你就完成了训练集与测试集的构建，并创建了一个 mnist 的网络模型。下面分别用两种方式完成模型的训练与预测。

2.5.2 二、通过 paddle.Model 训练与预测

你可以使用 paddle.Model 完成模型的封装，将网络结构组合成一个可快速使用高层 API 进行训练和预测的对象。代码如下：

```
model = paddle.Model(mnist)
```

2.1 用 Model.prepare() 配置模型

用 paddle.Model 完成模型的封装后，在训练前，需要对模型进行配置，通过 Model.prepare 接口来对训练进行提前的配置准备工作，包括设置模型优化器，Loss 计算方法，精度计算方法等。

```
# 为模型训练做准备，设置优化器，损失函数和精度计算方式
model.prepare(optimizer=paddle.optimizer.Adam(parameters=model.parameters()),
              loss=paddle.nn.CrossEntropyLoss(),
              metrics=paddle.metric.Accuracy())
```

2.2 用 Model.fit() 训练模型

做好模型训练的前期准备工作后，调用 fit() 接口来启动训练过程，需要指定至少 3 个关键参数：训练数据集，训练轮次和单次训练数据批次大小。

```
# 启动模型训练，指定训练数据集，设置训练轮次，设置每次数据集计算的批次大小，设置日志格式
model.fit(train_dataset,
          epochs=5,
          batch_size=64,
          verbose=1)
```

```
The loss value printed in the log is the current step, and the metric is the average␣
↪value of previous step.
Epoch 1/5
step 938/938 [==============================] - loss: 0.1785 - acc: 0.9281 - 19ms/step
Epoch 2/5
step 938/938 [==============================] - loss: 0.0365 - acc: 0.9688 - 19ms/step
Epoch 3/5
step 938/938 [==============================] - loss: 0.0757 - acc: 0.9781 - 19ms/step
Epoch 4/5
step 938/938 [==============================] - loss: 0.0054 - acc: 0.9824 - 19ms/step
Epoch 5/5
step 938/938 [==============================] - loss: 0.0640 - acc: 0.9858 - 19ms/step
```

1.3 用 Model.evaluate() 评估模型

对于训练好的模型进行评估可以使用 evaluate 接口，事先定义好用于评估使用的数据集后，直接调用 evaluate 接口即可完成模型评估操作，结束后根据在 prepare 中 loss 和 metric 的定义来进行相关评估结果计算返回。

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}

```
# 用 evaluate 在测试集上对模型进行验证
eval_result = model.evaluate(test_dataset, verbose=1)
```

```
Eval begin...
The loss value printed in the log is the current batch, and the metric is the average␣
↪value of previous step.
step 10000/10000 [==============================] - loss: 3.5763e-07 - acc: 0.9809 -␣
↪2ms/step
Eval samples: 10000
```

1.4 用 Model.predict() 预测模型

高层 API 中提供了 predict 接口来方便用户对训练好的模型进行预测验证，只需要基于训练好的模型将需要进行预测测试的数据放到接口中进行计算即可，接口会将经过模型计算得到的预测结果进行返回。

返回格式是一个 list，元素数目对应模型的输出数目：* 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), …]

numpy_ndarray_n 是对应原始数据经过模型计算后得到的预测数据，数目对应预测数据集的数目。

```
# 用 predict 在测试集上对模型进行测试
test_result = model.predict(test_dataset)
```

```
Predict begin...
step 10000/10000 [==============================] - 2ms/step
Predict samples: 10000
```

2.5.3 三、通过基础 API 实现模型的训练与预测

除了通过第一部分的高层 API 实现模型的训练与预测，飞桨框架也同样支持通过基础 API 对模型进行训练与预测。简单来说，`Model.prepare()`、`Model.fit()`、`Model.evaluate()`、`Model.predict()` 都是由基础 API 封装而来。下面通过拆解高层 API 到基础 API 的方式，来了解如何用基础 API 完成模型的训练与预测。

3.1 拆解 `Model.prepare()`、`Model.fit()`-用基础 API 训练模型

飞桨框架通过基础 API 对模型进行训练与预测，对应第一部分的 `Model.prepare()` 与 `Model.fit()`：

```
# dataset 与 mnist 的定义与第一部分内容一致

# 用 DataLoader 实现数据加载
train_loader = paddle.io.DataLoader(train_dataset, batch_size=64, shuffle=True)

mnist.train()

# 设置迭代次数
epochs = 5

# 设置优化器
optim = paddle.optimizer.Adam(parameters=mnist.parameters())
# 设置损失函数
loss_fn = paddle.nn.CrossEntropyLoss()

for epoch in range(epochs):
    for batch_id, data in enumerate(train_loader()):

        x_data = data[0]            # 训练数据
        y_data = data[1]            # 训练数据标签
        predicts = mnist(x_data)    # 预测结果

        # 计算损失 等价于 prepare 中 loss 的设置
        loss = loss_fn(predicts, y_data)

        # 计算准确率 等价于 prepare 中 metrics 的设置
        acc = paddle.metric.accuracy(predicts, y_data)

        # 下面的反向传播、打印训练信息、更新参数、梯度清零都被封装到 Model.fit() 中

        # 反向传播
        loss.backward()
```

(下页继续)

(续上页)

```
        if (batch_id+1) % 900 == 0:
            print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,␣
↪batch_id+1, loss.numpy(), acc.numpy()))

        # 更新参数
        optim.step()

        # 梯度清零
        optim.clear_grad()
```

```
epoch: 0, batch_id: 900, loss is: [0.29550618], acc is: [0.90625]
epoch: 1, batch_id: 900, loss is: [0.05875912], acc is: [0.984375]
epoch: 2, batch_id: 900, loss is: [0.05824642], acc is: [0.96875]
epoch: 3, batch_id: 900, loss is: [0.02940615], acc is: [1.]
epoch: 4, batch_id: 900, loss is: [0.05713747], acc is: [0.984375]
```

3.2 拆解 Model.evaluate()-用基础 API 验证模型

飞桨框架通过基础 API 对模型进行验证，对应第一部分的 Model.evaluate():

```
# 加载测试数据集
test_loader = paddle.io.DataLoader(test_dataset, batch_size=64, drop_last=True)
loss_fn = paddle.nn.CrossEntropyLoss()

mnist.eval()

for batch_id, data in enumerate(test_loader()):

    x_data = data[0]            # 测试数据
    y_data = data[1]            # 测试数据标签
    predicts = mnist(x_data)    # 预测结果

    # 计算损失与精度
    loss = loss_fn(predicts, y_data)
    acc = paddle.metric.accuracy(predicts, y_data)

    # 打印信息
    if (batch_id+1) % 30 == 0:
        print("batch_id: {}, loss is: {}, acc is: {}".format(batch_id+1, loss.numpy(),
↪ acc.numpy()))
```

```
batch_id: 30, loss is: [0.15860887], acc is: [0.953125]
batch_id: 60, loss is: [0.21005578], acc is: [0.921875]
batch_id: 90, loss is: [0.0889321], acc is: [0.953125]
batch_id: 120, loss is: [0.00115552], acc is: [1.]
batch_id: 150, loss is: [0.12016675], acc is: [0.984375]
```

3.3 拆解 Model.predict()-用基础 API 测试模型

飞桨框架通过基础 API 对模型进行测试，对应第一部分的 Model.predict()：

```
# 加载测试数据集
test_loader = paddle.io.DataLoader(test_dataset, batch_size=64, drop_last=True)

mnist.eval()
for batch_id, data in enumerate(test_loader()):
    x_data = data[0]
    predicts = mnist(x_data)
    # 获取预测结果
print("predict finished")
```

```
predict finished
```

2.6 资源配置

飞桨框架 2.0 增加 paddle.distributed.spawn 函数来启动单机多卡训练，同时原有的 paddle.distributed.launch 的方式依然保留。

2.6.1 一、launch 启动

1.1 高层 API 场景

当调用 paddle.Model 高层 API 来实现训练时，想要启动单机多卡训练非常简单，代码不需要做任何修改，只需要在启动时增加一下参数-m paddle.distributed.launch。

```
# 单机单卡启动，默认使用第 0 号卡
$ python train.py

# 单机多卡启动，默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py
```

(下页继续)

(续上页)

```
# 单机多卡启动，设置当前使用的第 0 号和第 1 号卡
$ python -m paddle.distributed.launch --gpus='0,1' train.py

# 单机多卡启动，设置当前使用第 0 号和第 1 号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

1.2 基础 API 场景

如果使用基础 API 实现训练，想要启动单机多卡训练，需要对单机单卡的代码进行 3 处修改，具体如下：

```
import paddle
# 第 1 处改动 导入分布式训练所需的包
import paddle.distributed as dist

# 加载数据集
train_dataset = paddle.vision.datasets.MNIST(mode='train')
test_dataset = paddle.vision.datasets.MNIST(mode='test')

# 定义网络结构
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(1, -1),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)

# 第 2 处改动，初始化并行环境
dist.init_parallel_env()

# 用 DataLoader 实现数据加载
train_loader = paddle.io.DataLoader(train_dataset, batch_size=32, shuffle=True)

# 第 3 处改动，增加 paddle.DataParallel 封装
mnist = paddle.DataParallel(mnist)
mnist.train()

# 设置迭代次数
epochs = 5

# 设置优化器
optim = paddle.optimizer.Adam(parameters=model.parameters())
```

(下页继续)

(续上页)

```
for epoch in range(epochs):
    for batch_id, data in enumerate(train_loader()):

        x_data = data[0]            # 训练数据
        y_data = data[1]            # 训练数据标签
        predicts = mnist(x_data)    # 预测结果

        # 计算损失 等价于 prepare 中 loss 的设置
        loss = paddle.nn.functional.cross_entropy(predicts, y_data)

        # 计算准确率 等价于 prepare 中 metrics 的设置
        acc = paddle.metric.accuracy(predicts, y_data)

        # 下面的反向传播、打印训练信息、更新参数、梯度清零都被封装到 Model.fit() 中

        # 反向传播
        loss.backward()

        if (batch_id+1) % 1800 == 0:
            print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,␣
↪batch_id, loss.numpy(), acc.numpy()))

        # 更新参数
        optim.step()

        # 梯度清零
        optim.clear_grad()
```

修改完后保存文件，然后使用跟高层 API 相同的启动方式即可。**注意：**单卡训练不支持调用 init_parallel_env，请使用以下几种方式进行分布式训练。

```
# 单机多卡启动，默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py

# 单机多卡启动，设置当前使用的第 0 号和第 1 号卡
$ python -m paddle.distributed.launch --gpus '0,1' train.py

# 单机多卡启动，设置当前使用第 0 号和第 1 号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

2.6.2 二、spawn 启动

launch 方式启动训练，以文件为单位启动多进程，需要用户在启动时调用 `paddle.distributed.launch`，对于进程的管理要求较高。飞桨框架 2.0 版本增加了 `spawn` 启动方式，可以更好地控制进程，在日志打印、训练退出时更友好。使用示例如下：

```
from __future__ import print_function

import paddle
import paddle.nn as nn
import paddle.optimizer as opt
import paddle.distributed as dist

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear1 = nn.Linear(10, 10)
        self._linear2 = nn.Linear(10, 1)

    def forward(self, x):
        return self._linear2(self._linear1(x))

def train(print_result=False):

    # 1. 初始化并行训练环境
    dist.init_parallel_env()

    # 2. 创建并行训练 Layer 和 Optimizer
    layer = LinearNet()
    dp_layer = paddle.DataParallel(layer)

    loss_fn = nn.MSELoss()
    adam = opt.Adam(
        learning_rate=0.001, parameters=dp_layer.parameters())

    # 3. 运行网络
    inputs = paddle.randn([10, 10], 'float32')
    outputs = dp_layer(inputs)
    labels = paddle.randn([10, 1], 'float32')
    loss = loss_fn(outputs, labels)

    if print_result is True:
        print("loss:", loss.numpy())
```

(下页继续)

(续上页)

```
    loss.backward()

    adam.step()
    adam.clear_grad()

# 使用方式 1: 仅传入训练函数
# 适用场景: 训练函数不需要任何参数，并且需要使用所有当前可见的 GPU 设备并行训练
if __name__ == '__main__':
    dist.spawn(train)

# 使用方式 2: 传入训练函数和参数
# 适用场景: 训练函数需要一些参数，并且需要使用所有当前可见的 GPU 设备并行训练
if __name__ == '__main__':
    dist.spawn(train, args=(True,))

# 使用方式 3: 传入训练函数、参数并指定并行进程数
# 适用场景: 训练函数需要一些参数，并且仅需要使用部分可见的 GPU 设备并行训练，例如:
# 当前机器有 8 张 GPU 卡 {0,1,2,3,4,5,6,7}，此时会使用前两张卡 {0,1};
# 或者当前机器通过配置环境变量 CUDA_VISIBLE_DEVICES=4,5,6,7，仅使 4 张
# GPU 卡可见，此时会使用可见的前两张卡 {4,5}
if __name__ == '__main__':
    dist.spawn(train, args=(True,), nprocs=2)

# 使用方式 4: 传入训练函数、参数、指定进程数并指定当前使用的卡号
# 使用场景: 训练函数需要一些参数，并且仅需要使用部分可见的 GPU 设备并行训练，但是
# 可能由于权限问题，无权配置当前机器的环境变量，例如: 当前机器有 8 张 GPU 卡
# {0,1,2,3,4,5,6,7}，但你无权配置 CUDA_VISIBLE_DEVICES，此时可以通过
# 指定参数 gpus 选择希望使用的卡，例如 gpus='4,5',
# 可以指定使用第 4 号卡和第 5 号卡
if __name__ == '__main__':
    dist.spawn(train, nprocs=2, gpus='4,5')
```

2.7 自定义指标

除了使用飞桨框架内置的指标外，飞桨框架还支持用户根据自己的实际场景，完成指标的自定义。

2.7.1 一、自定义 Loss

有时你会遇到特定任务的 Loss 计算方式在框架既有的 Loss 接口中不存在，或算法不符合自己的需求，那么期望能够自己来进行 Loss 的自定义。这里介绍如何进行 Loss 的自定义操作，首先来看下面的代码：

```
class SelfDefineLoss(paddle.nn.Layer):
    """
    1. 继承 paddle.nn.Layer
    """
    def __init__(self):
        """
        2. 构造函数根据自己的实际算法需求和使用需求进行参数定义即可
        """
        super(SelfDefineLoss, self).__init__()

    def forward(self, input, label):
        """
        3. 实现 forward 函数，forward 在调用时会传递两个参数：input 和 label
            - input：单个或批次训练数据经过模型前向计算输出结果
            - label：单个或批次训练数据对应的标签数据
            接口返回值是一个 Tensor，根据自定义的逻辑加和或计算均值后的损失
        """
        # 使用 Paddle 中相关 API 自定义的计算逻辑
        # output = xxxxx
        # return output
```

接下来是一个具体的例子，在图像分割示例代码中写的一个自定义 Loss，当时主要是使用自定义的 softmax 计算维度。

```
class SoftmaxWithCrossEntropy(paddle.nn.Layer):
    def __init__(self):
        super(SoftmaxWithCrossEntropy, self).__init__()

    def forward(self, input, label):
        loss = F.softmax_with_cross_entropy(input,
                                            label,
                                            return_softmax=False,
                                            axis=1)
        return paddle.mean(loss)
```

2.7.2 二、自定义 Metric

和 Loss 一样，你也可以来通过框架实现自定义的评估方法，具体的实现如下：

```
class SelfDefineMetric(paddle.metric.Metric):
    """
    1. 继承 paddle.metric.Metric
    """
    def __init__(self):
        """
        2. 构造函数实现，自定义参数即可
        """
        super(SelfDefineMetric, self).__init__()

    def name(self):
        """
        3. 实现 name 方法，返回定义的评估指标名字
        """
        return ' 自定义评价指标的名字'

    def compute(self, ...)
        """
        4. 本步骤可以省略，实现 compute 方法，这个方法主要用于 `update`的加速，可以在这个方法中调用
一些 paddle 实现好的 Tensor 计算 API，编译到模型网络中一起使用低层 C++ OP 计算。
        """
        return 自己想要返回的数据，会做为 update 的参数传入。

    def update(self, ...):
        """
        5. 实现 update 方法，用于单个 batch 训练时进行评估指标计算。
        - 当 `compute`类函数未实现时，会将模型的计算输出和标签数据的展平作为 `update`的参数传入。
        - 当 `compute`类函数做了实现时，会将 compute 的返回结果作为 `update`的参数传入。
        """
        return acc value

    def accumulate(self):
        """
        6. 实现 accumulate 方法，返回历史 batch 训练积累后计算得到的评价指标值。
        每次 `update`调用时进行数据积累，`accumulate`计算时对积累的所有数据进行计算并返回。
        结算结果会在 `fit`接口的训练日志中呈现。
        """
        # 利用 update 中积累的成员变量数据进行计算后返回
        return accumulated acc value

    def reset(self):
```

(下页继续)

(续上页)

```
        """
        7. 实现 reset 方法，每个 Epoch 结束后进行评估指标的重置，这样下个 Epoch 可以重新进行计算。
        """
        # do reset action
```

接下来看一个框架中的具体例子，是框架中已提供的一个评估指标计算接口，这里就是按照上述说明中的方法完成了实现。

```
from paddle.metric import Metric

class Precision(Metric):
    """
    Precision (also called positive predictive value) is the fraction of
    relevant instances among the retrieved instances. Refer to
    https://en.wikipedia.org/wiki/Evaluation_of_binary_classifiers
    Noted that this class manages the precision score only for binary
    classification task.

    ......
    """

    def __init__(self, name='precision', *args, **kwargs):
        super(Precision, self).__init__(*args, **kwargs)
        self.tp = 0  # true positive
        self.fp = 0  # false positive
        self._name = name

    def update(self, preds, labels):
        """
        Update the states based on the current mini-batch prediction results.
        Args:
            preds (numpy.ndarray): The prediction result, usually the output
                of two-class sigmoid function. It should be a vector (column
                vector or row vector) with data type: 'float64' or 'float32'.
           labels (numpy.ndarray): The ground truth (labels),
                the shape should keep the same as preds.
                The data type is 'int32' or 'int64'.
        """
        if isinstance(preds, paddle.Tensor):
            preds = preds.numpy()
        elif not _is_numpy_(preds):
            raise ValueError("The 'preds' must be a numpy ndarray or Tensor.")
        if isinstance(labels, paddle.Tensor):
```

(下页继续)

(续上页)

```
            labels = labels.numpy()
        elif not _is_numpy_(labels):
            raise ValueError("The 'labels' must be a numpy ndarray or Tensor.")

        sample_num = labels.shape[0]
        preds = np.floor(preds + 0.5).astype("int32")

        for i in range(sample_num):
            pred = preds[i]
            label = labels[i]
            if pred == 1:
                if pred == label:
                    self.tp += 1
                else:
                    self.fp += 1

    def reset(self):
        """
        Resets all of the metric state.
        """
        self.tp = 0
        self.fp = 0

    def accumulate(self):
        """
        Calculate the final precision.

        Returns:
           A scaler float: results of the calculated precision.
        """
        ap = self.tp + self.fp
        return float(self.tp) / ap if ap != 0 else .0

    def name(self):
        """
        Returns metric name
        """
        return self._name
```

2.7.3 三、自定义 Callback

fit 接口的 callback 参数支持传入一个 " Callback"类实例，用来在每轮训练和每个 " batch"训练前后进行调用，可以通过 " callback"收集到训练过程中的一些数据和参数，或者实现一些自定义操作。

```
class SelfDefineCallback(paddle.callbacks.Callback):
    """
    1. 继承 paddle.callbacks.Callback
    2. 按照自己的需求实现以下类成员方法:
        def on_train_begin(self, logs=None)                 训练开始前，`Model.fit`接口中
调用
        def on_train_end(self, logs=None)                   训练结束后，`Model.fit`接口中
调用
        def on_eval_begin(self, logs=None)                  评估开始前，`Model.
↪evaluate`接口调用
        def on_eval_end(self, logs=None)                    评估结束后，`Model.
↪evaluate`接口调用
        def on_predict_begin(self, logs=None)               预测测试开始前，`Model.
↪predict`接口中调用
        def on_predict_end(self, logs=None)                 预测测试结束后，`Model.
↪predict`接口中调用
        def on_epoch_begin(self, epoch, logs=None)          每轮训练开始前，`Model.fit`接
口中调用
        def on_epoch_end(self, epoch, logs=None)            每轮训练结束后，`Model.fit`接
口中调用
        def on_train_batch_begin(self, step, logs=None)     单个 Batch 训练开始前，
`Model.fit`和 `Model.train_batch`接口中调用
        def on_train_batch_end(self, step, logs=None)       单个 Batch 训练结束后，
`Model.fit`和 `Model.train_batch`接口中调用
        def on_eval_batch_begin(self, step, logs=None)      单个 Batch 评估开始前，
`Model.evalute`和 `Model.eval_batch`接口中调用
        def on_eval_batch_end(self, step, logs=None)        单个 Batch 评估结束后，
`Model.evalute`和 `Model.eval_batch`接口中调用
        def on_predict_batch_begin(self, step, logs=None)   单个 Batch 预测测试开始前，
`Model.predict`和 `Model.test_batch`接口中调用
        def on_predict_batch_end(self, step, logs=None)     单个 Batch 预测测试结束后，
`Model.predict`和 `Model.test_batch`接口中调用
    """

    def __init__(self):
        super(SelfDefineCallback, self).__init__()
    # 按照需求定义自己的类成员方法
```

看一个框架中的实际例子，这是框架自带的 " ModelCheckpoint"回调函数，可以在 " fit"训练模型时自动存储每轮训练得到的模型。

```
class ModelCheckpoint(Callback):
    def __init__(self, save_freq=1, save_dir=None):
        self.save_freq = save_freq
        self.save_dir = save_dir

    def on_epoch_begin(self, epoch=None, logs=None):
        self.epoch = epoch

    def _is_save(self):
        return self.model and self.save_dir and ParallelEnv().local_rank == 0

    def on_epoch_end(self, epoch, logs=None):
        if self._is_save() and self.epoch % self.save_freq == 0:
            path = '{}/{}'.format(self.save_dir, epoch)
            print('save checkpoint at {}'.format(os.path.abspath(path)))
            self.model.save(path)

    def on_train_end(self, logs=None):
        if self._is_save():
            path = '{}/final'.format(self.save_dir)
            print('save checkpoint at {}'.format(os.path.abspath(path)))
            self.model.save(path)
```

2.8 模型存储与载入

2.8.1 一、存储载入体系简介

1.1 接口体系

飞桨框架 2.x 对模型与参数的存储与载入相关接口进行了梳理，根据接口使用的场景与模式，分为三套体系，分别是：

1.1.1 动态图存储载入体系

为提升框架使用体验，飞桨框架 2.0 将主推动态图模式，动态图模式下的存储载入接口包括：

- paddle.save
- paddle.load
- paddle.jit.save
- paddle.jit.load

本文主要介绍飞桨框架 2.0 动态图存储载入体系，各接口关系如下图所示：

1.1.2 静态图存储载入体系

静态图存储载入相关接口为飞桨框架 1.x 版本的主要使用接口，出于兼容性的目的，这些接口仍然可以在飞桨框架 2.x 使用，但不再推荐。相关接口包括：

- paddle.static.save
- paddle.static.load
- paddle.static.save_inference_model
- paddle.static.load_inference_model
- paddle.static.load_program_state
- paddle.static.set_program_state

由于飞桨框架 2.0 不再主推静态图模式，故本文不对以上主要用于飞桨框架 1.x 的相关接口展开介绍，如有需要，可以阅读对应 API 文档。

1.1.3 高阶 API 存储载入体系

- paddle.Model.fit (训练接口，同时带有参数存储的功能)
- paddle.Model.save
- paddle.Model.load

飞桨框架 2.0 高阶 API 仅有一套 Save/Load 接口，表意直观，体系清晰，若有需要，建议直接阅读相关 API 文档，此处不再赘述。

注解： 本教程着重介绍飞桨框架 2.x 的各个存储载入接口的关系及各种使用场景，不对接口参数进行详细介绍，如果需要了解具体接口参数的含义，请直接阅读对应 API 文档。

1.2 接口存储结果组织形式

飞桨 2.0 统一了各存储接口对于同一种存储行为的处理方式，并且统一推荐或自动为存储的文件添加飞桨标准的文件后缀，详见下图：

2.8.2 二、参数存储载入（训练调优）

若仅需要存储/载入模型的参数，可以使用 `paddle.save/load` 结合 Layer 和 Optimizer 的 state_dict 达成目的，此处 state_dict 是对象的持久参数的载体，dict 的 key 为参数名，value 为参数真实的 numpy array 值。

结合以下简单示例，介绍参数存储和载入的方法，以下示例完成了一个简单网络的训练过程：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
```

(下页继续)

(续上页)

```
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

# create network
layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# train
train(layer, loader, loss_fn, adam)
```

2.1 参数存储

参数存储时，先获取目标对象（Layer 或者 Optimzier）的 state_dict，然后将 state_dict 存储至磁盘，示例如下（接前述示例）：

```
# save
paddle.save(layer.state_dict(), "linear_net.pdparams")
paddle.save(adam.state_dict(), "adam.pdopt")
```

2.2 参数载入

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，示例如下（接前述示例）：

```
# load
layer_state_dict = paddle.load("linear_net.pdparams")
opt_state_dict = paddle.load("adam.pdopt")

layer.set_state_dict(layer_state_dict)
adam.set_state_dict(opt_state_dict)
```

2.8.3 三、模型 & 参数存储载入（训练部署）

若要同时存储/载入模型结构和参数，可以使用 `paddle.jit.save/load` 实现。

3.1 模型 & 参数存储

模型 & 参数存储根据训练模式不同，有两种使用情况：

(1) 动转静训练 + 模型 & 参数存储

(2) 动态图训练 + 模型 & 参数存储

3.1.1 动转静训练 + 模型 & 参数存储

动转静训练相比直接使用动态图训练具有更好的执行性能，训练完成后，直接将目标 Layer 传入 `paddle.jit.save` 存储即可。：

一个简单的网络训练示例如下：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
```

(下页继续)

(续上页)

```
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

# create network
layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# train
train(layer, loader, loss_fn, adam)
```

随后使用 `paddle.jit.save` 对模型和参数进行存储（接前述示例）：

```
# save
path = "example.model/linear"
paddle.jit.save(layer, path)
```

通过动转静训练后保存模型 & 参数，有以下三项注意点：

(1) Layer 对象的 forward 方法需要经由 `paddle.jit.to_static` 装饰

经过 paddle.jit.to_static 装饰 forward 方法后，相应 Layer 在执行时，会先生成描述模型的 Program，然后通过执行 Program 获取计算结果，示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)
```

若最终需要生成的描述模型的 Program 支持动态输入，可以同时指明模型的 InputSepc ，示例如下：

```
import paddle
import paddle.nn as nn
from paddle.static import InputSpec

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
    def forward(self, x):
        return self._linear(x)
```

(2) 请确保 Layer.forward 方法中仅实现预测功能，避免将训练所需的 loss 计算逻辑写入 forward 方法

Layer 更准确的语义是描述一个具有预测功能的模型对象，接收输入的样本数据，输出预测的结果，而 loss 计算是仅属于模型训练中的概念。将 loss 计算的实现放到 Layer.forward 方法中，会使 Layer 在不同场景下概念有所差别，并且增大 Layer 使用的复杂性，这不是良好的编码行为，同时也会在最终保存预测模型时引入剪枝的复杂性，因此建议保持 Layer 实现的简洁性，下面通过两个示例对比说明：

错误示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x, label=None):
        out = self._linear(x)
        if label:
            loss = nn.functional.cross_entropy(out, label)
            avg_loss = nn.functional.mean(loss)
            return out, avg_loss
        else:
            return out
```

正确示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)
```

(3) 如果你需要存储多个方法，需要用 `paddle.jit.to_static` 装饰每一个需要被存储的方法。

注解： 只有在 forward 之外还需要存储其他方法时才用这个特性，如果仅装饰非 forward 的方法，而 forward 没有被装饰，是不符合规范的。此时 `paddle.jit.save` 的 `input_spec` 参数必须为 None。

示例代码如下：

```
import paddle
import paddle.nn as nn
from paddle.static import InputSpec

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)
        self._linear_2 = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, IMAGE_SIZE], dtype=
↪'float32')])
    def forward(self, x):
        return self._linear(x)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, IMAGE_SIZE], dtype=
↪'float32')])
    def another_forward(self, x):
        return self._linear_2(x)

inps = paddle.randn([1, IMAGE_SIZE])
layer = LinearNet()
before_0 = layer.another_forward(inps)
before_1 = layer(inps)
# save and load
path = "example.model/linear"
paddle.jit.save(layer, path)
```

存储的模型命名规则：forward 的模型名字为：模型名 + 后缀，其他函数的模型名字为：模型名 + 函数名 + 后缀。每个函数有各自的 pdmodel 和 pdiparams 的文件，所有函数共用 pdiparams.info。上述代码将在 example.model 文件夹下产生 5 个文件：linear.another_forward.pdiparams、linear.pdiparams、linear.pdmodel、linear.another_forward.pdmodel、linear.pdiparams.info

3.1.2 动态图训练 + 模型 & 参数存储

动态图模式相比动转静模式更加便于调试，如果你仍需要使用动态图直接训练，也可以在动态图训练完成后调用 `paddle.jit.save` 直接存储模型和参数。

同样是一个简单的网络训练示例：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt
from paddle.static import InputSpec

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
```

(下页继续)

(续上页)

```
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

# create network
layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# train
train(layer, loader, loss_fn, adam)
```

训练完成后使用 `paddle.jit.save` 对模型和参数进行存储：

```
# save
path = "example.dy_model/linear"
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
```

动态图训练后使用 `paddle.jit.save` 存储模型和参数注意点如下：

(1) 相比动转静训练，Layer 对象的 forward 方法不需要额外装饰，保持原实现即可

(2) 与动转静训练相同，请确保 Layer.forward 方法中仅实现预测功能，避免将训练所需的 loss 计算逻辑写入 forward 方法

(3) 在最后使用 `paddle.jit.save` 时，需要指定 Layer 的 `InputSpec` ，Layer 对象 forward 方法的每一个参数均需要对应的 `InputSpec` 进行描述，不能省略。这里的 `input_spec` 参数支持两种类型的输入：

- `InputSpec` 列表

使用 InputSpec 描述 forward 输入参数的 shape，dtype 和 name，如前述示例（此处示例中 name 省略，name 省

略的情况下会使用 forward 的对应参数名作为 name，所以这里的 name 为 x ）：

```
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
```

- Example Tensor 列表

除使用 InputSpec 之外，也可以直接使用 forward 训练时的示例输入，此处可以使用前述示例中迭代 DataLoader 得到的 image ，示例如下：

```
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[image])
```

3.2 模型 & 参数载入

载入模型参数，使用 paddle.jit.load 载入即可，载入后得到的是一个 Layer 的派生类对象 TranslatedLayer ，TranslatedLayer 具有 Layer 具有的通用特征，支持切换 train 或者 eval 模式，可以进行模型调优或者预测。

注解： 为了规避变量名字冲突，载入之后会重命名变量。

载入模型及参数，示例如下：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# load
path = "example.model/linear"
loaded_layer = paddle.jit.load(path)
```

载入模型及参数后进行预测，示例如下（接前述示例）：

```
# inference
loaded_layer.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = loaded_layer(x)
```

载入模型及参数后进行调优，示例如下（接前述示例）：

```
# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

# fine-tune
loaded_layer.train()
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=loaded_layer.parameters())
train(loaded_layer, loader, loss_fn, adam)
# save after fine-tuning
paddle.jit.save(loaded_layer, "fine-tune.model/linear", input_spec=[x])
```

此外，paddle.jit.save 同时保存了模型和参数，如果你只需要从存储结果中载入模型的参数，可以使用 paddle.load 接口载入，返回所存储模型的 state_dict，示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)

# create network
layer = LinearNet()

# load
path = "example.model/linear"
state_dict = paddle.load(path)

# inference
layer.set_state_dict(state_dict, use_structured_name=False)
layer.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = layer(x)
```

2.8.4 四、旧存储格式兼容载入

如果你是从飞桨框架 1.x 切换到 2.x，曾经使用飞桨框架 1.x 的 fluid 相关接口存储模型或者参数，飞桨框架 2.x 也对这种情况进行了兼容性支持，包括以下几种情况。

飞桨 1.x 模型准备及训练示例，该示例为后续所有示例的前序逻辑：

```
import numpy as np
import paddle
import paddle.fluid as fluid
import paddle.nn as nn
import paddle.optimizer as opt
```

(下页继续)

(续上页)

```
BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# enable static mode
paddle.enable_static()

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

image = fluid.data(name='image', shape=[None, 784], dtype='float32')
label = fluid.data(name='label', shape=[None, 1], dtype='int64')
pred = fluid.layers.fc(input=image, size=10, act='softmax')
loss = fluid.layers.cross_entropy(input=pred, label=label)
avg_loss = fluid.layers.mean(loss)

optimizer = fluid.optimizer.SGD(learning_rate=0.001)
optimizer.minimize(avg_loss)

place = fluid.CPUPlace()
exe = fluid.Executor(place)
exe.run(fluid.default_startup_program())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    feed_list=[image, label],
    places=place,
    batch_size=BATCH_SIZE,
    shuffle=True,
```

(下页继续)

(续上页)

```
    drop_last=True,
    num_workers=2)

# train model
for data in loader():
    exe.run(
        fluid.default_main_program(),
        feed=data,
        fetch_list=[avg_loss])
```

4.1 从 paddle.fluid.io.save_inference_model 存储结果中载入模型 & 参数

(1) 同时载入模型和参数

使用 paddle.jit.load 配合 **configs 载入模型和参数。

如果你是按照 paddle.fluid.io.save_inference_model 的默认格式存储的，可以按照如下方式载入（接前述示例）：

```
# save default
model_path = "fc.example.model"
fluid.io.save_inference_model(
    model_path, ["image"], [pred], exe)

# enable dynamic mode
paddle.disable_static(place)

# load
fc = paddle.jit.load(model_path)

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)
```

如果你指定了存储的模型文件名，可以按照以下方式载入（接前述示例）：

```
# save with model_filename
model_path = "fc.example.model.with_model_filename"
fluid.io.save_inference_model(
    model_path, ["image"], [pred], exe, model_filename="__simplenet__")

# enable dynamic mode
paddle.disable_static(place)
```

(下页继续)

(续上页)

```
# load
fc = paddle.jit.load(model_path, model_filename="__simplenet__")

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)
```

如果你指定了存储的参数文件名，可以按照以下方式载入（接前述示例）：

```
# save with params_filename
model_path = "fc.example.model.with_params_filename"
fluid.io.save_inference_model(
    model_path, ["image"], [pred], exe, params_filename="__params__")

# enable dynamic mode
paddle.disable_static(place)

# load
fc = paddle.jit.load(model_path, params_filename="__params__")

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)
```

(2) 仅载入参数

如果你仅需要从 `paddle.fluid.io.save_inference_model` 的存储结果中载入参数，以 state_dict 的形式配置到已有代码的模型中，可以使用 `paddle.load` 配合 `**configs` 载入。

如果你是按照 `paddle.fluid.io.save_inference_model` 的默认格式存储的，可以按照如下方式载入（接前述示例）：

```
model_path = "fc.example.model"

load_param_dict = paddle.load(model_path)
```

如果你指定了存储的模型文件名，可以按照以下方式载入（接前述示例）：

```
model_path = "fc.example.model.with_model_filename"

load_param_dict = paddle.load(model_path, model_filename="__simplenet__")
```

如果你指定了存储的参数文件名，可以按照以下方式载入（接前述示例）：

```
model_path = "fc.example.model.with_params_filename"

load_param_dict = paddle.load(model_path, params_filename="__params__")
```

注解： 一般预测模型不会存储优化器 Optimizer 的参数，因此此处载入的仅包括模型本身的参数。

注解： 由于 `structured_name` 是动态图下独有的变量命名方式，因此从静态图存储结果载入的 state_dict 在配置到动态图的 Layer 中时，需要配置 `Layer.set_state_dict(use_structured_name=False)` 。

4.2 从 `paddle.fluid.save` 存储结果中载入参数

`paddle.fluid.save` 的存储格式与 2.x 动态图接口 `paddle.save` 存储格式是类似的，同样存储了 dict 格式的参数，因此可以直接使用 `paddle.load` 载入 state_dict，但需要注意不能仅传入保存的路径，而要传入保存参数的文件名，示例如下（接前述示例）：

```
# save by fluid.save
model_path = "fc.example.model.save"
program = fluid.default_main_program()
fluid.save(program, model_path)

# enable dynamic mode
paddle.disable_static(place)

load_param_dict = paddle.load("fc.example.model.save.pdparams")
```

注解： 由于 `paddle.fluid.save` 接口原先在静态图模式下的定位是存储训练时参数，或者说存储 Checkpoint，故尽管其同时存储了模型结构，目前也暂不支持从 `paddle.fluid.save` 的存储结果中同时载入模型和参数，后续如有需求再考虑支持。

4.3 从 `paddle.fluid.io.save_params/save_persistables` 存储结果中载入参数

这两个接口在飞桨 1.x 版本时，已经不再推荐作为存储模型参数的接口使用，故并未继承至飞桨 2.x，之后也不会再推荐使用这两个接口存储参数。

对于使用这两个接口存储参数兼容载入的支持，分为两种情况，下面以 `paddle.fluid.io.save_params` 接口为例介绍相关使用方法：

(1) 使用默认方式存储，各参数分散存储为单独的文件，文件名为参数名

这种存储方式仍然可以使用 paddle.load 接口兼容载入，使用示例如下（接前述示例）：

```
# save by fluid.io.save_params
model_path = "fc.example.model.save_params"
fluid.io.save_params(exe, model_path)

# load
state_dict = paddle.load(model_path)
print(state_dict)
```

(2) 指定了参数存储的文件，将所有参数存储至单个文件中

将所有参数存储至单个文件中会导致存储结果中丢失 Tensor 名和 Tensor 数据之间的映射关系，因此这部分丢失的信息需要用户传入进行补足。为了确保正确性，这里不仅要传入 Tensor 的 name 列表，同时要传入 Tensor 的 shape 和 dtype 等描述信息，通过检查和存储数据的匹配性确保严格的正确性，这导致载入数据的恢复过程变得比较复杂，仍然需要一些飞桨 1.x 的概念支持。后续如果此项需求较为普遍，飞桨将会考虑将该项功能兼容支持到 paddle.load 中，但由于信息丢失而导致的使用复杂性仍然是存在的，因此建议你避免仅使用这两个接口存储参数。

目前暂时推荐你使用 paddle.static.load_program_state 接口解决此处的载入问题，需要获取原 Program 中的参数列表传入该方法，使用示例如下（接前述示例）：

```
# save by fluid.io.save_params
model_path = "fc.example.model.save_params_with_filename"
fluid.io.save_params(exe, model_path, filename="__params__")

# load
import os
params_file_path = os.path.join(model_path, "__params__")
var_list = fluid.default_main_program().all_parameters()
state_dict = paddle.io.load_program_state(params_file_path, var_list)
```

2.9 模型导出 ONNX 协议

2.9.1 一、简介

ONNX (Open Neural Network Exchange) 是针对机器学习所设计的开源文件格式，用于存储训练好的模型。它使得不同的人工智能框架可以采用相同格式存储模型并交互。通过 ONNX 格式，Paddle 模型可以使用 OpenVINO、ONNX Runtime 等框架进行推理。

Paddle 转 ONNX 协议由 paddle2onnx 实现，下面介绍如何将 Paddle 模型转换为 ONNX 模型并验证正确性。

本教程涉及的示例代码，可点击 IPython 获取，除 Paddle 以外，还需安装以下依赖：

```
pip install paddle2onnx onnx onnxruntime // -i https://mirror.baidu.com/pypi/simple 如果网速不好，可以使用其他源下载
```

2.9.2 二、模型导出为 ONNX 协议

2.1 动态图导出 ONNX 协议

Paddle 动态图模型转换为 ONNX 协议，首先会将 Paddle 的动态图 `paddle.nn.Layer` 转换为静态图，详细原理可以参考 动态图转静态图 。然后依照 ONNX 的算子协议，将 Paddle 的算子一一映射为 ONNX 的算子。动态图转换 ONNX 调用 `paddle.onnx.export()` 接口即可实现，该接口通过 `input_spec` 参数为模型指定输入的形状和数据类型，支持 `Tensor` 或 `InputSpec` ，其中 `InputSpec` 支持动态的 shape。

关于 `paddle.onnx.export` 接口更详细的使用方法，请参考 API 。

```
import paddle
from paddle import nn
from paddle.static import InputSpec

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(784, 10)

    def forward(self, x):
        return self._linear(x)

# export to ONNX
layer = LinearNet()
save_path = 'onnx.save/linear_net'
x_spec = InputSpec([None, 784], 'float32', 'x')
paddle.onnx.export(layer, save_path, input_spec=[x_spec])
```

2.2 静态图导出 ONNX 协议

Paddle 2.0 以后将主推动态图组网方式，如果您的模型来自于旧版本的 Paddle，使用静态图组网，请参考 paddle2onnx 的 使用文档 和 示例 。

2.9.3 三、ONNX 模型的验证

ONNX 官方工具包提供了 API 可验证模型的正确性，主要包括两个方面，一是算子是否符合对应版本的协议，二是网络结构是否完整。

```
# check by ONNX
import onnx

onnx_file = save_path +  '.onnx'
onnx_model = onnx.load(onnx_file)
onnx.checker.check_model(onnx_model)
print('The model is checked!')
```

如果模型检查失败，请到 Paddle 或 paddle2onnx 提出 Issue，我们会跟进相应的问题。

2.9.4 四、ONNXRuntime 推理

本节介绍使用 ONNXRuntime 对已转换的 Paddle 模型进行推理，并与使用 Paddle 进行推理的结果进行对比。

```
import numpy as np
import onnxruntime

x = np.random.random((2, 784)).astype('float32')

# predict by ONNX Runtime
ort_sess = onnxruntime.InferenceSession(onnx_file)
ort_inputs = {ort_sess.get_inputs()[0].name: x}
ort_outs = ort_sess.run(None, ort_inputs)

print("Exported model has been predicted by ONNXRuntime!")

# predict by Paddle
layer.eval()
paddle_outs = layer(x)

# compare ONNX Runtime and Paddle results
np.testing.assert_allclose(ort_outs[0], paddle_outs.numpy(), rtol=1.0, atol=1e-05)

print("The difference of results between ONNXRuntime and Paddle looks good!")
```

2.9.5 五、相关链接

- 算子转换支持列表
- 模型转换支持列表

CHAPTER 3

VisualDL 工具

3.1 VisualDL 工具简介

VisualDL 是飞桨可视化分析工具，以丰富的图表呈现训练参数变化趋势、模型结构、数据样本、直方图、PR 曲线及高维数据分布。可帮助用户更清晰直观地理解深度学习模型训练过程及模型结构，进而实现高效的模型优化。

具体功能使用方式请参见 **VisualDL 使用指南**。项目正处于高速迭代中，敬请期待新组件的加入。

VisualDL 支持浏览器种类：Chrome（81 和 83）、Safari 13、FireFox（77 和 78）、Edge（Chromium 版）。

VisualDL 原生支持 python 的使用，通过在模型的 Python 配置中添加几行代码，便可为训练过程提供丰富的可视化支持。

3.1.1 目录

- 核心亮点
- 安装方式
- 使用方式
- 可视化功能概览
- 开源贡献
- 更多细节
- 技术交流

3.1.2 核心亮点

简单易用

API 设计简洁易懂，使用简单。模型结构一键实现可视化。

功能丰富

功能覆盖标量、数据样本、图结构、直方图、PR 曲线及数据降维可视化。

高兼容性

全面支持 Paddle、ONNX、Caffe 等市面主流模型结构可视化，广泛支持各类用户进行可视化分析。

全面支持

与飞桨服务平台及工具组件全面打通，为您在飞桨生态系统中提供最佳使用体验。

3.1.3 安装方式

使用 pip 安装

```
pip install --upgrade --pre visualdl
```

使用代码安装

```
git clone https://github.com/PaddlePaddle/VisualDL.git
cd VisualDL

python setup.py bdist_wheel
pip install --upgrade dist/visualdl-*.whl
```

需要注意，官方自 2020 年 1 月 1 日起不再维护 Python2，为了保障代码可用性，VisualDL 现仅支持 Python3

3.1.4 使用方式

VisualDL 将训练过程中的数据、参数等信息储存至日志文件中后，启动面板即可查看可视化结果。

1. 记录日志

VisualDL 的后端提供了 Python SDK，可通过 LogWriter 定制一个日志记录器，接口如下：

```
class LogWriter(logdir=None,
                comment='',
                max_queue=10,
                flush_secs=120,
                filename_suffix='',
                write_to_disk=True,
                **kwargs)
```

接口参数

示例

设置日志文件并记录标量数据：

```
from visualdl import LogWriter

# 在 `./log/scalar_test/train`路径下建立日志文件
with LogWriter(logdir="./log/scalar_test/train") as writer:
    # 使用 scalar 组件记录一个标量数据
    writer.add_scalar(tag="acc", step=1, value=0.5678)
    writer.add_scalar(tag="acc", step=2, value=0.6878)
    writer.add_scalar(tag="acc", step=3, value=0.9878)
```

2. 启动面板

在上述示例中，日志已记录三组标量数据，现可启动 VisualDL 面板查看日志的可视化结果，共有两种启动方式：

在命令行启动

使用命令行启动 VisualDL 面板，命令格式如下：

```
visualdl --logdir <dir_1, dir_2, ... , dir_n> --host <host> --port <port> --cache-
↪timeout <cache_timeout> --language <language> --public-path <public_path> --api-only
```

参数详情：

针对上一步生成的日志，启动命令为：

```
visualdl --logdir ./log
```

在 Python 脚本中启动

支持在 Python 脚本中启动 VisualDL 面板，接口如下：

```
visualdl.server.app.run(logdir,
                        host="127.0.0.1",
                        port=8080,
                        cache_timeout=20,
                        language=None,
                        public_path=None,
                        api_only=False,
                        open_browser=False)
```

请注意：除 `logdir` 外，其他参数均为不定参数，传递时请指明参数名。

接口参数具体如下：

针对上一步生成的日志，我们的启动脚本为：

```
from visualdl.server import app

app.run(logdir="./log")
```

在使用任意一种方式启动 VisualDL 面板后，打开浏览器访问 VisualDL 面板，即可查看日志的可视化结果，如图：

3.1.5 可视化功能概览

Scalar

以图表形式实时展示训练过程参数，如 loss、accuracy。让用户通过观察单组或多组训练参数变化，了解训练过程，加速模型调优。具有两大特点：

动态展示

在启动 VisualDL 后，LogReader 将不断增量的读取日志中数据并供前端调用展示，因此能够在训练中同步观测指标变化，如下图：

多实验对比

只需在启动 VisualDL 时将每个实验日志所在路径同时传入即可，每个实验中相同 tag 的指标将绘制在一张图中同步呈现，如下图：

Image

实时展示训练过程中的图像数据，用于观察不同训练阶段的图像变化，进而深入了解训练过程及效果。

Audio

实时查看训练过程中的音频数据，监控语音识别与合成等任务的训练过程。

Graph

一键可视化模型的网络结构。可查看模型属性、节点信息、节点输入输出等，并支持节点搜索，辅助用户快速分析模型结构与了解数据流向。

Histogram

以直方图形式展示 Tensor（weight、bias、gradient 等）数据在训练过程中的变化趋势。深入了解模型各层效果，帮助开发者精准调整模型结构。

- Offset 模式
- Overlay 模式

PR Curve

精度-召回率曲线，帮助开发者权衡模型精度和召回率之间的平衡，设定最佳阈值。

High Dimensional

将高维数据进行降维展示，目前支持 T-SNE、PCA 两种降维方式，用于深入分析高维数据间的关系，方便用户根据数据特征进行算法优化。

3.1.6 开源贡献

VisualDL 是由 PaddlePaddle 和 ECharts 合作推出的开源项目。Graph 相关功能由 Netron 提供技术支持。欢迎所有人使用，提意见以及贡献代码。

3.1.7 更多细节

想了解更多关于 VisualDL 可视化功能的使用详情介绍，请查看 **VisualDL 使用指南**。

3.1.8 技术交流

欢迎您加入 VisualDL 官方 QQ 群：1045783368 与飞桨团队以及其他用户共同针对 VisualDL 进行讨论与交流。

3.2 VisualDL 使用指南

3.2.1 概述

VisualDL 是一个面向深度学习任务设计的可视化工具。VisualDL 利用了丰富的图表来展示数据，用户可以更直观、清晰地查看数据的特征与变化趋势，有助于分析数据、及时发现错误，进而改进神经网络模型的设计。

目前，VisualDL 支持 scalar, image, audio, graph, histogram, pr curve, high dimensional 七个组件，项目正处于高速迭代中，敬请期待新组件的加入。

3.2.2 Scalar –折线图组件

介绍

Scalar 组件的输入数据类型为标量，该组件的作用是将训练参数以折线图形式呈现。将损失函数值、准确率等标量数据作为参数传入 scalar 组件，即可画出折线图，便于观察变化趋势。

记录接口

Scalar 组件的记录接口如下：

```
add_scalar(tag, value, step, walltime=None)
```

接口参数说明如下：

Demo

- 基础使用

下面展示了使用 Scalar 组件记录数据的示例，代码文件请见Scalar 组件

```
from visualdl import LogWriter

if __name__ == '__main__':
    value = [i/1000.0 for i in range(1000)]
    # 初始化一个记录器
    with LogWriter(logdir="./log/scalar_test/train") as writer:
        for step in range(1000):
            # 向记录器添加一个 tag 为 `acc`的数据
            writer.add_scalar(tag="acc", step=step, value=value[step])
            # 向记录器添加一个 tag 为 `loss`的数据
            writer.add_scalar(tag="loss", step=step, value=1/(value[step] + 1))
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 `http://127.0.0.1:8080`，即可查看以下折线图。

- 多组实验对比

下面展示了使用 Scalar 组件实现多组实验对比

多组实验对比的实现分为两步：

1. 创建子日志文件储存每组实验的参数数据
2. 将数据写入 scalar 组件时，**使用相同的 tag**，即可实现对比**不同实验**的**同一类型参数**

```
from visualdl import LogWriter

if __name__ == '__main__':
    value = [i/1000.0 for i in range(1000)]
    # 步骤一：创建父文件夹：log 与子文件夹：scalar_test
    with LogWriter(logdir="./log/scalar_test") as writer:
```

(下页继续)

(续上页)

```
        for step in range(1000):
            # 步骤二: 向记录器添加一个 tag 为 `train/acc`的数据
            writer.add_scalar(tag="train/acc", step=step, value=value[step])
            # 步骤二: 向记录器添加一个 tag 为 `train/loss`的数据
            writer.add_scalar(tag="train/loss", step=step, value=1/(value[step] + 1))
    # 步骤一: 创建第二个子文件夹 scalar_test2
    value = [i/500.0 for i in range(1000)]
    with LogWriter(logdir="./log/scalar_test2") as writer:
        for step in range(1000):
            # 步骤二: 在同样名为 `train/acc`下添加 scalar_test2 的 accuracy 的数据
            writer.add_scalar(tag="train/acc", step=step, value=value[step])
            # 步骤二: 在同样名为 `train/loss`下添加 scalar_test2 的 loss 的数据
            writer.add_scalar(tag="train/loss", step=step, value=1/(value[step] + 1))
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 `http://127.0.0.1:8080`，即可查看以下折线图，对比「scalar_test」和「scalar_test2」的 Accuracy 和 Loss。

* 多组实验对比的应用案例可参考 AI Studio 项目：VisualDL 2.0–眼疾识别训练可视化

功能操作说明

- 支持数据卡片「最大化」、「还原」、「坐标系转化」(y 轴对数坐标)、「下载」折线图
- 数据点 Hover 展示详细信息
- 可搜索卡片标签，展示目标图像
- 可搜索打点数据标签，展示特定数据
- X 轴有三种衡量尺度

1. Step：迭代次数
2. Walltime：训练绝对时间
3. Relative：训练时长

- 可调整曲线平滑度，以便更好的展现参数整体的变化趋势

3.2.3 Image –图片可视化组件

介绍

Image 组件用于显示图片数据随训练的变化。在模型训练过程中，将图片数据传入 Image 组件，就可在 VisualDL 的前端网页查看相应图片。

记录接口

Image 组件的记录接口如下：

```
add_image(tag, img, step, walltime=None)
```

接口参数说明如下：

Demo

下面展示了使用 Image 组件记录数据的示例，代码文件请见Image 组件

```
import numpy as np
from PIL import Image
from visualdl import LogWriter


def random_crop(img):
    """ 获取图片的随机 100x100 分片
    """
    img = Image.open(img)
    w, h = img.size
    random_w = np.random.randint(0, w - 100)
    random_h = np.random.randint(0, h - 100)
    r = img.crop((random_w, random_h, random_w + 100, random_h + 100))
    return np.asarray(r)


if __name__ == '__main__':
    # 初始化一个记录器
    with LogWriter(logdir="./log/image_test/train") as writer:
        for step in range(6):
            # 添加一个图片数据
            writer.add_image(tag="eye",
                             img=random_crop("../../docs/images/eye.jpg"),
                             step=step)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 http://127.0.0.1:8080，即可查看图片数据。

功能操作说明

可搜索图片标签显示对应图片数据

支持滑动 Step/迭代次数查看不同迭代次数下的图片数据

3.2.4 Audio–音频播放组件

介绍

Audio 组件实时查看训练过程中的音频数据，监控语音识别与合成等任务的训练过程。

记录接口

Audio 组件的记录接口如下：

```
add_audio(tag, audio_array, step, sample_rate)
```

接口参数说明如下：

Demo

下面展示了使用 Audio 组件记录数据的示例，代码文件请见Audio 组件

```
from visualdl import LogWriter
import numpy as np
import wave


def read_audio_data(audio_path):
    """
    Get audio data.
    """
    CHUNK = 4096
    f = wave.open(audio_path, "rb")
    wavdata = []
    chunk = f.readframes(CHUNK)
```

(下页继续)

(续上页)

```
    while chunk:
        data = np.frombuffer(chunk, dtype='uint8')
        wavdata.extend(data)
        chunk = f.readframes(CHUNK)
    # 8k sample rate, 16bit frame, 1 channel
    shape = [8000, 2, 1]
    return shape, wavdata


if __name__ == '__main__':
    with LogWriter(logdir="./log") as writer:
        audio_shape, audio_data = read_audio_data("./testing.wav")
        audio_data = np.array(audio_data)
        writer.add_audio(tag="audio_tag",
                         audio_array=audio_data,
                         step=0,
                         sample_rate=8000)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 http://127.0.0.1:8080，即可查看音频数据。

功能操作说明

- 可搜索音频标签显示对应音频数据
- 支持滑动 Step/迭代次数试听不同迭代次数下的音频数据
- 支持播放/暂停音频数据
- 支持音量调节
- 支持音频下载

3.2.5 Graph-网络结构组件

介绍

Graph 组件一键可视化模型的网络结构。用于查看模型属性、节点信息、节点输入输出等，并进行节点搜索，协助开发者们快速分析模型结构与了解数据流向。

Demo

共有两种启动方式：

- 前端模型文件拖拽上传：
  - 如只需使用 Graph 组件，则无需添加任何参数，在命令行执行 `visualdl` 后即可启动面板进行上传。
  - 如果同时需使用其他功能，在命令行指定日志文件路径（以`./log`为例）即可启动面板进行上传：

```
visualdl --logdir ./log --port 8080
```

- 后端启动 Graph：
  - 在命令行加入参数`--model` 并指定**模型文件**路径（非文件夹路径），即可启动并查看网络结构可视化：

```
visualdl --model ./log/model --port 8080
```

功能操作说明

- 一键上传模型
  - 支持模型格式：PaddlePaddle、ONNX、Keras、Core ML、Caffe、Caffe2、Darknet、MXNet、ncnn、TensorFlow Lite
  - 实验性支持模型格式：TorchScript、PyTorch、Torch、ArmNN、BigDL、Chainer、CNTK、Deeplearning4j、MediaPipe、ML.NET、MNN、OpenVINO、Scikit-learn、Tengine、TensorFlow.js、TensorFlow
- 支持上下左右任意拖拽模型、放大和缩小模型
- 搜索定位到对应节点
- 点击查看模型属性
- 支持选择模型展示的信息
- 支持以 PNG、SVG 格式导出模型结构图
- 点击节点即可展示对应属性信息
- 支持一键更换模型

3.2.6 Histogram–直方图组件

介绍

Histogram 组件以直方图形式展示 Tensor（weight、bias、gradient 等）数据在训练过程中的变化趋势。深入了解模型各层效果，帮助开发者精准调整模型结构。

记录接口

Histogram 组件的记录接口如下：

```
add_histogram(tag, values, step, walltime=None, buckets=10)
```

接口参数说明如下：

Demo

下面展示了使用 Histogram 组件记录数据的示例，代码文件请见Histogram 组件

```
from visualdl import LogWriter
import numpy as np


if __name__ == '__main__':
    values = np.arange(0, 1000)
    with LogWriter(logdir="./log/histogram_test/train") as writer:
        for index in range(1, 101):
            interval_start = 1 + 2 * index / 100.0
            interval_end = 6 - 2 * index / 100.0
            data = np.random.uniform(interval_start, interval_end, size=(10000))
            writer.add_histogram(tag='default tag',
                                 values=data,
                                 step=index,
                                 buckets=10)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 http://127.0.0.1:8080，即可查看训练参数直方图。

功能操作说明

- 支持数据卡片「最大化」、直方图「下载」
- 可选择 Offset 或 Overlay 模式
  - Offset 模式
  - Overlay 模式
- 数据点 Hover 展示参数值、训练步数、频次
  - 在第 240 次训练步数时，权重为-0.0031，且出现的频次是 2734 次
- 可搜索卡片标签，展示目标直方图
- 可搜索打点数据标签，展示特定数据流

3.2.7 PR Curve–PR 曲线组件

介绍

PR Curve 以折线图形式呈现精度与召回率的权衡分析，清晰直观了解模型训练效果，便于分析模型是否达到理想标准。

记录接口

PR Curve 组件的记录接口如下：

```
add_pr_curve(tag, labels, predictions, step=None, num_thresholds=10)
```

接口参数说明如下：

Demo

下面展示了使用 PR Curve 组件记录数据的示例，代码文件请见 *PR Curve* 组件

```
from visualdl import LogWriter
import numpy as np

with LogWriter("./log/pr_curve_test/train") as writer:
    for step in range(3):
        labels = np.random.randint(2, size=100)
        predictions = np.random.rand(100)
        writer.add_pr_curve(tag='pr_curve',
                            labels=labels,
```

(下页继续)

(续上页)

```
                                predictions=predictions,
                                step=step,
                                num_thresholds=5)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 http://127.0.0.1:8080，即可查看 PR Curve

功能操作说明

- 支持数据卡片「最大化」,「还原」、「下载」PR 曲线
- 数据点 Hover 展示详细信息：阈值对应的 TP、TN、FP、FN
- 可搜索卡片标签，展示目标图表
- 可搜索打点数据标签，展示特定数据
- 支持查看不同训练步数下的 PR 曲线
- X 轴-时间显示类型有三种衡量尺度
  - Step：迭代次数
  - Walltime：训练绝对时间
  - Relative：训练时长

3.2.8 High Dimensional –数据降维组件

介绍

High Dimensional 组件将高维数据进行降维展示，用于深入分析高维数据间的关系。目前支持以下两种降维算法：

- PCA : Principle Component Analysis 主成分分析
- t-SNE : t-distributed stochastic neighbor embedding t-分布式随机领域嵌入

记录接口

High Dimensional 组件的记录接口如下：

```
add_embeddings(tag, labels, hot_vectors, walltime=None)
```

接口参数说明如下：

Demo

下面展示了使用 High Dimensional 组件记录数据的示例，代码文件请见High Dimensional 组件

```
from visualdl import LogWriter


if __name__ == '__main__':
    hot_vectors = [
        [1.3561076367500755, 1.3116267195134017, 1.6785401875616097],
        [1.1039614644440658, 1.8891609992484688, 1.32030488587171],
        [1.9924524852447711, 1.9358920727142739, 1.2124401279391606],
        [1.4129542689796446, 1.7372166387197474, 1.7317806077076527],
        [1.3913371800587777, 1.4684674577930312, 1.5214136352476377]]

    labels = ["label_1", "label_2", "label_3", "label_4", "label_5"]
    # 初始化一个记录器
    with LogWriter(logdir="./log/high_dimensional_test/train") as writer:
        # 将一组 labels 和对应的 hot_vectors 传入记录器进行记录
        writer.add_embeddings(tag='default',
                              labels=labels,
                              hot_vectors=hot_vectors)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 http://127.0.0.1:8080，即可查看降维后的可视化数据。

CHAPTER 4

动态图转静态图

动态图有诸多优点，包括易用的接口，Python 风格的编程体验，友好的 debug 交互机制等。在动态图模式下，代码是按照我们编写的顺序依次执行。这种机制更符合 Python 程序员的习惯，可以很方便地将大脑中的想法快速地转化为实际代码，也更容易调试。但在性能方面，Python 执行开销较大，与 C++ 有一定差距。因此在工业界的许多部署场景中（如大型推荐系统、移动端）都倾向于直接使用 C++ 来提速。

相比动态图，静态图在部署方面更具有性能的优势。静态图程序在编译执行时，先搭建模型的神经网络结构，然后再对神经网络执行计算操作。预先搭建好的神经网络可以脱离 Python 依赖，在 C++ 端被重新解析执行，而且拥有整体网络结构也能进行一些网络结构的优化。

动态图代码更易编写和 debug，但在部署性能上，静态图更具优势。因此我们新增了动态图转静态图的功能，支持用户依然使用动态图编写组网代码。PaddlePaddle 会对用户代码进行分析，自动转换为静态图网络结构，兼顾了动态图易用性和静态图部署性能两方面优势。

我们在以下链接介绍 PaddlePaddle 动态图转静态图的各个部分：

- 基本用法：介绍了动态图转静态图的基本使用方法
- 内部架构原理： 介绍了动态图转静态图的架构原理
- 支持语法列表： 介绍了动态图转静态图支持的语法以及罗列不支持的语法写法
- InputSpec 功能介绍： 介绍了动态图转静态图指定输入 InputSpec 的功能和用法
- 报错信息处理： 介绍了动态图转静态图的报错信息处理方法
- 调试方法： 介绍了动态图转静态图支持的调试方法

4.1 基本用法

PaddlePaddle 主要的动转静方式是基于源代码级别转换的 ProgramTranslator。其基本原理是通过分析 Python 代码来将动态图代码转写为静态图代码，并在底层自动帮用户使用静态图执行器运行。这种转换方式使得用户可以灵活使用 Python 语法及其控制流来构建神经网络模型。除此之外，PaddlePaddle 另外提供一种基于 trace 的动转静接口 TracedLayer。若遇到 ProgramTranslator 不支持但是可以用 TracedLayer 运行的情况，可以作为备选方案。

4.1.1 基于源代码转写的 ProgramTranslator

源代码转写的 ProgramTranslator 进行动态图转静态图，其基本原理是通过分析 Python 代码来将动态图代码转写为静态图代码，并在底层自动帮用户使用执行器运行。其基本使用方法十分简便，只需要在要转化的函数（该函数也可以是用户自定义动态图 Layer 的 forward 函数）前添加一个装饰器 `@paddle.jit.to_static`，一个转化例子如下，可以直接运行被装饰函数得到结果：

```
import paddle

@paddle.jit.to_static
def func(input_var):
    # if 判断与输入 input_var 的 shape 有关
    if input_var.shape[0] > 1:
        out = paddle.cast(input_var, "float64")
    else:
        out = paddle.cast(input_var, "int64")
    return out

in_np = np.array([-2]).astype('int')
input_var = paddle.to_tensor(in_np)
func(input_var)
```

若要存储转化后的静态图模型，可以调用 `paddle.jit.save`，我们定义一个简单全连接网络 SimpleFcLayer，需要在下面 SimpleFcLayer 的 forward 函数添加装饰器：

```
import numpy as np
import paddle

class SimpleFcLayer(paddle.nn.Layer):
    def __init__(self, batch_size, feature_size, fc_size):
        super(SimpleFcLayer, self).__init__()
        self._linear = paddle.nn.Linear(feature_size, fc_size)
        self._offset = paddle.to_tensor(
            np.random.random((batch_size, fc_size)).astype('float32'))
```

(下页继续)

(续上页)

```
    @paddle.jit.to_static
    def forward(self, x):
        fc = self._linear(x)
        return fc + self._offset
```

存储该模型可以使用 paddle.jit.save 接口：

```
import paddle

fc_layer = SimpleFcLayer(3, 4, 2)
in_np = np.random.random([3, 4]).astype('float32')
input_var = paddle.to_tensor(in_np)
out = fc_layer(input_var)

paddle.jit.save(fc_layer, "./fc_layer_dy2stat", input_spec=[input_var])
```

4.1.2 基于 trace 的 TracedLayer

trace 是指在模型运行时记录下其运行过哪些算子。TracedLayer 就是基于这种技术，在一次执行动态图的过程中，记录所有运行的算子，并构建和保存静态图模型。一个使用例子如下：

我们还是定义一个简单的全连接网络作为例子，注意这里不需要像 ProgramTranslator 在 forward 函数添加装饰器：

```
import numpy as np
import paddle

class SimpleFcLayer(paddle.nn.Layer):
    def __init__(self, batch_size, feature_size, fc_size):
        super(SimpleFcLayer, self).__init__()
        self._linear = paddle.nn.Linear(feature_size, fc_size)
        self._offset = paddle.to_tensor(
            np.random.random((batch_size, fc_size)).astype('float32'))

    def forward(self, x):
        fc = self._linear(x)
        return fc + self._offset
```

接下来是 TracedLayer 如何存储模型：

```
import paddle
from paddle.jit import TracedLayer

```

(下页继续)

(续上页)

```
fc_layer = SimpleFcLayer(3, 4, 2)
in_np = np.random.random([3, 4]).astype('float32')
# 将 numpy 的 ndarray 类型的数据转换为 Tensor 类型
input_var = paddle.to_tensor(in_np)
# 通过 TracerLayer.trace 接口将命令式模型转换为声明式模型
out_dygraph, static_layer = TracedLayer.trace(fc_layer, inputs=[input_var])
save_dirname = './saved_infer_model'
# 将转换后的模型保存
static_layer.save_inference_model(save_dirname, feed=[0], fetch=[0])
```

载入的模型可以使用静态图方式运行

```
place = paddle.CPUPlace()
exe = paddle.Executor(place)
program, feed_vars, fetch_vars = paddle.static.load_inference_model(save_dirname, exe)
fetch, = exe.run(program, feed={feed_vars[0]: in_np}, fetch_list=fetch_vars)
```

但是也正如我们阐述的原理，trace 只是记录了一次执行涉及的算子。若在用户的模型代码中，包含了依赖数据条件（包括输入的值或者 shape）的控制流分支，即根据数据条件触发运行不同的算子，则 TracedLayer 无法正常工作。比如下面：

```
import paddle

def func(input_var)
    # if 判断与输入 input_var 的 shape 有关
    if input_var.shape[0] > 1:
        return paddle.cast(input_var, "float64")
    else:
        return paddle.cast(input_var, "int64")

in_np = np.array([-2]).astype('int')
input_var = paddle.to_tensor(in_np)
out = func(input_var)
```

如果对上述样例中的 `func` 使用 `TracedLayer.trace(func, inputs=[input_var])` ，由于 trace 只能记录 if-else 其中跑的一次算子，模型就无法按用户想要的根据 input_var 的形状进行 if-else 控制流保存。类似的控制流还有 while/for 循环的情况。

4.1.3 比较 ProgramTranslator 和 TracedLayer

基于源代码转换的 ProgramTranslator 对比基于 trace 的 TracedLayer，前者能够处理依赖数据条件的控制流分支。因此我们更推荐用户使用 ProgramTranslator，如果遇到问题再以 TracedLayer 作为备选方案。

4.2 内部架构原理

TracedLayer 的原理就是 trace，相对简单，因此我们在这里不展开描述。本节将主要阐述 ProgramTranslator 基于源代码将动态图代码转化为静态图代码。

转化过程发生在用户开始调用被装饰的函数，转换过程在装饰器中实现。我们将内部涉及的过程分为以下几步：

4.2.1 函数与缓存

动态图转静态图的主体是函数（Function）。对于函数内包含的 PaddlePaddle 接口，如果是仅计算相关算子代码语句，那么因为 PaddlePaddle 动态图和静态图接口一致，我们不需要额外转换这些代码为静态图代码。但是对于动态图，此类代码接口是直接运行计算和返回结果，而对于静态图此类代码接口其实是组网。那么如果被转化的函数被调用多次，动态图转静态图后会多次组网添加对应算子，这显然会导致问题。为了解决这个问题以及为了加速动转静转化过程，我们维护了被装饰器装饰的函数（Function）与其输入形状（shape），数据类型（dtype）映射到被转化后组网的 Program 的缓存（Cache）。当要被转化的函数命中缓存，我们直接用对应存储的 Program 运行静态图得到结果，否则我们才进行语句转化，并且转化成功后的 Program 存储进缓存。

4.2.2 动态图源码转 AST（抽象语法树）

动态图转静态图的最核心部分类似一个编译器，解析动态图代码语句为 AST，再对应 AST 进行改写，最后反转回成静态图代码。从函数转化为代码字符串可以使用 Python 的 inspect.getsource。从字符串 Python 提供了自带的 ast 库来解析字符串为 AST，但是由于 Python2，Python3 的语法略有不同，为了避免我们需要额外处理这些 Python2，Python3 的不同情况，我们使用了统一 Python2，Python3 的开源 AST 处理 gast 库 。这些接口使得函数转化为 AST 没有本质上的困难。

4.2.3 AST 改写和静态图源码转换

这部分为动转静最核心的部分，我们对支持的各种语法进行 ast 转写。其中最重要的 Python 控制流，if-else，while，for 循环被分别分析转化为 PaddlePaddle 静态图接口 cond，while_loop 等接口实现。我们对想转化的每一种主要语法创建一个 Transformer（这里的 Transformer 是 Python ast 转写的概念，而不是自然语言处理 NLP 领域的 Transformer），每个 Transformer 扫一遍 AST 并进行对应的改写。最后被转化完成的 AST 我们使用 gast 提供的接口转回成源码。

4.2.4 静态图源码作为动态图一部分运行的技术

为了动静转化更加易用和被转化的代码能在动态图中复用，我们在拥有源码后运行生成 Program，并将这个 Program 作为一个大 op，包装成动态图的一个 op，这样既能把用户的代码转为静态图提速或者保存部署，另一方面如果用户想在 Python 层使用生成的静态图代码作为动态图的一部分继续训练或者别的动态图运算也是可以直接使用。

4.2.5 易用性与 Debug 功能在动转静过程的实现

正如 AST 转写类似编译器，而一般编译器都会提供 debug 断点，报错，输出一些中间代码等功能。我们在进行动转静时，万一用户的动态图代码出错，或者用户想断点调试，或者用户想看看被转化后的静态图代码是否符合其预期，我们也希望能够像编译器一样提供这些易用性功能，使得动转静兼顾性能和部署同时还具有易用性。我们这里将列出这些功能的实现方式

A. 报错对应到动态图代码行。由于被转化后的静态图代码和原动态图代码不同，Python 运行出错时会报静态图的错误，因此我们在每一次 AST 转写时添加 AST 节点对应的原动态图代码行等信息，在 Python 报错栈中将静态图的报错转化成对应的动态图源码报错

B. 设置断点功能。我们保留了被转化后代码的中的 pdb.set_trace(), 用户可以使用这种方式进行断点调试

C. 查看最后转化的静态图代码。我们输出为一个 StaticLayer class，这个 StaticLayer 可以直接被调用，但是也存储转化后的代码，可以调用 StaticLayer.code 来获得转化后的代码。

D. 输出中间转化状态代码，甚至不同语法 Transformer 转化的代码，比如经过 for 循环转化后代码是什么样的。我们开放接口设定了 log level 来让用户可以打印中间状态转化的代码。

4.3 支持语法列表

ProgramTranslator 本质是把 Python 运行语法转写为 PaddlePaddle 静态图代码，但是 Python 语法的表达能力和 PaddlePaddle 静态图表达能力存在不同，这使得一些代码无法被转换。

本章节我们将详细讲述在动转静过程中支持转化哪些语法，不支持哪些语法，并且讲述如何改写代码能够解决语法不支持的场景。

动转静支持的语法分为以下几个大类：

4.3.1 控制流相关关键词

控制流指 if-elif-else，while 等能够控制程序语句执行顺序的关键字。PaddlePaddle 静态图通过 cond，while_loop API 来实现条件判断和循环，如果动态图 Python 控制流的判断条件或循环条件依赖 PaddlePaddle Tensor，动转静后会被转化为等价的 PaddlePaddle 控制流接口，否则仍然使用 Python 控制流逻辑运行。在动转静过程中这些关键字的转化情况为：

1. if-elif-else 条件

当 `if < 条件>` 中的条件是 Tensor 时，ProgramTranslator 会把该 if-elif-else 语句转化为等价的 cond API 语句。否则会按普通 Python if-elif-else 的逻辑运行。需注意 cond 支持的 Tensor 只能是 numel 为 1 的 bool Tensor，所以请使用这种 Tensor 进行条件判断，其他 Tensor 会报错。

2. while 循环

当 while 循环中的条件是 Tensor 时，ProgramTranslator 会把该 while 语句转化为等价的 while_loop API 语句，否则会按普通 Python while 运行。需注意 while 循环条件中的 Tensor 只能是 numel 为 1 的 bool Tensor，所以请使用这种 Tensor 进行条件判断，其他 Tensor 会报错。

3. for 循环

3.1 `for _ in range(__)` 循环

ProgramTranslator 先将其转化为等价的 Python while 循环，然后按 while 循环的逻辑进行动静转换。

3.2 `for _ in x` 循环

当 x 是 Python 容器或迭代器，则会用普通 Python 逻辑运行。当 x 是 Tensor 时，会转化为循环中每次对应拿出 x[0], x[1], ···。

3.3 `for idx, val in enumerate(x)` 循环

当 x 是 Python 容器或迭代器，则会用普通 Python 逻辑运行。当 x 是 Tensor 时，idx 会转化为依次 0，1，···的 1-D Tensor。val 会转化为循环中每次对应拿出 x[0], x[1], ···。

4. break，continue

ProgramTranslator 可以支持在循环中添加 break，continue 语句，其底层实现原理是对于要 break，continue 的部分在相应时候使用 cond 在一定条件下跳过执行。

5. return

ProgramTranslator 支持在循环，条件判断中 return 结果而不需要一定在函数末尾 return。也能够支持 return 不同长度 tuple 和不同类型的 Tensor。其底层实现原理是对 return 后的部分相应使用 cond 在一定条件下跳过执行。

4.3.2 一些需要转化的运算类型

1. +，-，，/，*, >, <, >= , <=, == 等 Python 内置运算

由于静态图有重载这些基本运算符，所以这些被 ProgramTranslator 转化后都适用相应重载的运算符，动转静支持此类运算。

2. and，or，not 逻辑运算

Python 内置 and，or，not 逻辑运算关键词，ProgramTranslator 在语句的运算时会判断逻辑运算关键词运行的对象是否是 Tensor，如果都是 Tensor，我们将其转化为静态图对应的逻辑运算接口并运行。

3. 类型转化

动态图中可以直接用 Python 的类型转化语法来转化 Tensor 类型。例如 x 是 Tensor 时，float(x) 可以将 x 的类型转化为 float。ProgramTranslator 在运行时判断 x 是否是 Tensor，如果是，则在动转静时使用静态图 cast 接口转化相应的 Tensor 类型。

4.3.3 Python 函数相关

1. print

如果 x 是 Tensor，在动态图模式中 print(x) 可以打印 x 的值。在动转静过程中我们把此转化为静态图的 Print 接口实现，使得在静态图中也能打印。如果 print 的参数不是 Tensor，那么我们没有把相应 print 语句进行转写。

2. len

如果 x 是 Tensor，在动态图模式中 len(x) 可以获得 x 第 0 维度的长度。在动转静中我们把此转化为静态图 shape 接口，并返回 shape 的第 0 维。另外如果 x 是个 TensorArray，那么 len(x) 将会使用静态图接口 control_flow.array_length 返回 TensorArray 的长度。对于其他情况，动转静时会按照普通 Python len 函数运行。

3. lambda 表达式

动转静允许写带有 Python lambda 表达式的语句，并且我们会适当改写使得返回对应结果。

4. 函数内再调用函数

对于函数内调用其他函数的情况，ProgramTranslator 也会对内部的函数递归地进行动转静，这样做的好处是可以在最外层函数只需加一次装饰器即可，而不需要每个函数都加装饰器。但需要注意，动转静还不支持函数递归调用自己，详细原因请查看下文动转静无法正确运行的情况。

4.3.4 报错异常相关

1. assert

如果 x 是 Tensor，在动态图中可以通过 assert x 来强制 x 为 True 或者非 0 值，在动转静中我们把此转化为静态图 Assert 接口支持此功能。

4.3.5 Python 基本容器

1. list：对于一个 list 如果里面元素都是 Tensor，那么动转静会转化其为 TensorArray，静态图 TensorArray 可以支持 append，pop，修改操作。因此 ProgramTranslator 在元素皆为 Tensor 的 list 中支持上面三种操作。换言之，其他 list 操作，比如 sort 无法支持。对于 list 中并非所有元素是 Tensor 的情况，ProgramTranslator 会将其作为普通 Python list 运行。
2. dict：ProgramTranslator 会将相应的 dict 中的 Tensor 添加进静态图 Program，因此使用 dict 是动转静支持的语法。

4.3.6 动转静无法正确运行的情况

1. 改变变量的 shape 后调用其 shape 作为 PaddlePaddle API 参数。

具体表现比如 `x = reshape(x, shape=shape_tensor)`，再使用 `x.shape[0]` 的值进行其他操作。这种情况会由于动态图和静态图的本质不同而使得动态图能够运行，但静态图运行失败。其原因是动态图情况下，API 是直接返回运行结果，因此 `x.shape` 在经过 reshape 运算后是确定的。但是在转化为静态图后，因为静态图 API 只是组网，`shape_tensor` 的值在组网时是不知道的，所以 `reshape` 接口组网完，静态图并不知道 `x.shape` 的值。PaddlePaddle 静态图用-1 表示未知的 shape 值，此时 `x` 的 shape 每个维度会被设为-1，而不是期望的值。同理，类似 expand 等更改 shape 的 API，其输出 Tensor 再调用 shape 也难以进行动转静。

遇到这类情况我们建议尽量固定 shape 值，减少变化 shape 操作。

2. 多重 list 嵌套读写 Tensor

具体表现如 `l = [[tensor1, tensor2], [tensor3, tensor4]]`，因为现在动转静将元素全是 Tensor 的 list 转化为 TensorArray，而 PaddlePaddle 的 TensorArray 还不支持多维数组，因此这种情况下，动转静无法正确运行。

遇到这类情况我们建议尽量用一维 list，或者自己使用 PaddlePaddle 的 create_array，array_read，array_write 接口编写为 TensorArray。

3. Tensor 值在被装饰函数中转成 numpy array 进行运算

具体表现为在被装饰函数中没有返回 Tensor 时就使用 `numpy.array(tensor)` 将 Tensor 转化为 numpy array 并使用 numpy 接口进行运算。这种情况在动态图下因为 Tensor 有值是可以正常运行的，但是在静态图时由于 Tensor 只是组网变量，在没有运行时没有数值，因此无法进行 numpy 运算。

遇到这种情况我们建议在动转静的函数中尽量使用 PaddlePaddle 接口替代 numpy 接口进行运算。

4. 一个函数递归调用本身

ProgramTranslator 还无法支持一个函数递归调用本身，原因是递归常常会用 `if-else` 构造停止递归的条件。然而这样的停止条件在静态图下只是一个 `cond` 组网，组网并不能在编译阶段得到递归条件决定本身组多少次，会导致函数运行时一直组网递归直至栈溢出，因此 ProgramTranslator 还无法支持一个函数递归调用本身。

遇到这种情况我们建议将代码改为非递归写法。

4.4 InputSpec 功能介绍

在 PaddlePaddle（下文简称：Paddle）框架中，可以通过 `paddle.jit.to_static` 装饰普通函数或 Layer 的最外层 forward 函数，将动态图模型转换为静态图执行。但在动转静时，需要给模型传入 Tensor 数据并执行一次前向，以保证正确地推导出网络中各 Tensor 的 shape 。此转换流程需要显式地执行一次动态图函数，增加了接口使用的成本；同时，传入实际 Tensor 数据则无法定制化模型输入的 shape，如指定某些维度为 None 。

因此，Paddle 提供了 InputSpec 接口，可以更加便捷地执行动转静功能，以及定制化输入 Tensor 的 shape 、name 等信息。

4.4.1 一、InputSpec 对象构造方法

1.1 直接构造 InputSpec 对象

InputSpec 接口在 `paddle.static` 目录下，用于描述一个 Tensor 的签名信息：shape、dtype、name。使用样例如下：

```
from paddle.static import InputSpec

x = InputSpec([None, 784], 'float32', 'x')
label = InputSpec([None, 1], 'int64', 'label')

print(x)       # InputSpec(shape=(-1, 784), dtype=VarType.FP32, name=x)
print(label)   # InputSpec(shape=(-1, 1), dtype=VarType.INT64, name=label)
```

InputSpec 初始化中的只有 `shape` 是必须参数，`dtype` 和 `name` 可以缺省，默认取值分别为 `float32` 和 `None` 。

1.2 根据 Tensor 构造 InputSpec 对象

可以借助 `InputSpec.from_tensor` 方法，从一个 Tensor 直接创建 InputSpec 对象，其拥有与源 Tensor 相同的 `shape` 和 `dtype` 。使用样例如下：

```
import numpy as np
import paddle
from paddle.static import InputSpec

x = paddle.to_tensor(np.ones([2, 2], np.float32))
x_spec = InputSpec.from_tensor(x, name='x')
print(x_spec)  # InputSpec(shape=(2, 2), dtype=VarType.FP32, name=x)
```

注解： 若未在 `from_tensor` 中指定新的 name，则默认使用与源 Tensor 相同的 name。

1.3 根据 numpy.ndarray 构造 InputSpec 对象

也可以借助 `InputSpec.from_numpy` 方法，从一个 Numpy.ndarray 直接创建 InputSpec 对象，其拥有与源 ndarray 相同的 `shape` 和 `dtype` 。使用样例如下：

```
import numpy as np
from paddle.static import InputSpec

```

(下页继续)

(续上页)

```
x = np.ones([2, 2], np.float32)
x_spec = InputSpec.from_numpy(x, name='x')
print(x_spec)  # InputSpec(shape=(2, 2), dtype=VarType.FP32, name=x)
```

注解： 若未在 from_numpy 中指定新的 name，则默认使用 None 。

4.4.2 二、基本使用方法

动转静 paddle.jit.to_static 装饰器支持 input_spec 参数，用于指定被装饰函数每个 Tensor 类型输入参数的 shape 、dtype 、name 等签名信息。不必再显式地传入 Tensor 数据以触发网络层 shape 的推导。Paddle 会解析 to_static 中指定的 input_spec 参数，构建网络的起始输入，进行后续的模型组网。

同时，借助 input_spec 参数，可以自定义输入 Tensor 的 shape ，比如指定 shape 为 [None, 784] ，其中 None 表示变长的维度。

2.1 to_static 装饰器模式

如下是一个简单的使用样例：

```
import paddle
from paddle.jit import to_static
from paddle.static import InputSpec
from paddle.fluid.dygraph import Layer

class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), InputSpec(shape=[3],
↪ name='y')])
    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()

# save static model for inference directly
paddle.jit.save(net, './simple_net')
```

在上述的样例中，`to_static` 装饰器中的 `input_spec` 为一个 InputSpec 对象组成的列表，用于依次指定参数 x 和 y 对应的 Tensor 签名信息。在实例化 SimpleNet 后，可以直接调用 `paddle.jit.save` 保存静态图模型，不需要执行任何其他的代码。

注解:

1. input_spec 参数中只支持 InputSpec 对象，暂不支持如 int 、float 等类型。
2. 若指定 input_spec 参数，则需为被装饰函数的所有必选参数都添加对应的 InputSpec 对象，如上述样例中，不支持仅指定 x 的签名信息。
3. 若被装饰函数中包括非 Tensor 参数，且指定了 input_spec ，请确保函数的非 Tensor 参数都有默认值，如 `forward(self, x, use_bn=False)`

2.2 to_static 函数调用

若期望在动态图下训练模型，在训练完成后保存预测模型，并指定预测时需要的签名信息，则可以选择在保存模型时，直接调用 `to_static` 函数。使用样例如下：

```
class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()

# train process (Pseudo code)
for epoch_id in range(10):
    train_step(net, train_reader)

net = to_static(net, input_spec=[InputSpec(shape=[None, 10], name='x'),␣
↪InputSpec(shape=[3], name='y')])

# save static model for inference directly
paddle.jit.save(net, './simple_net')
```

如上述样例代码中，在完成训练后，可以借助 `to_static(net, input_spec=...)` 形式对模型实例进行处理。Paddle 会根据 input_spec 信息对 forward 函数进行递归的动转静，得到完整的静态图，且包括当前训练好的参数数据。

2.3 支持 list 和 dict 推导

上述两个样例中，被装饰的 forward 函数的参数均为 Tensor 。这种情况下，参数个数必须与 InputSpec 个数相同。但当被装饰的函数参数为 list 或 dict 类型时，`input_spec` 需要与函数参数保持相同的嵌套结构。

当函数的参数为 list 类型时，input_spec 列表中对应元素的位置，也必须是包含相同元素的 InputSpec 列表。使用样例如下：

```
class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[[InputSpec(shape=[None, 10], name='x'),␣
↪InputSpec(shape=[3], name='y')]])
    def forward(self, inputs):
        x, y = inputs[0], inputs[1]
        out = self.linear(x)
        out = out + y
        return out
```

其中 `input_spec` 参数是长度为 1 的 list ，对应 forward 函数的 inputs 参数。`input_spec[0]` 包含了两个 InputSpec 对象，对应于参数 inputs 的两个 Tensor 签名信息。

当函数的参数为 dict 时，`input_spec` 列表中对应元素的位置，也必须是包含相同键（key）的 InputSpec 列表。使用样例如下：

```
class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), {'x':␣
↪InputSpec(shape=[3], name='bias')}])
    def forward(self, x, bias_info):
        x_bias = bias_info['x']
        out = self.linear(x)
        out = out + x_bias
        return out
```

其中 `input_spec` 参数是长度为 2 的 list ，对应 forward 函数的 x 和 bias_info 两个参数。`input_spec` 的最后一个元素是包含键名为 x 的 InputSpec 对象的 dict ，对应参数 bias_info 的 Tensor 签名信息。

2.4 指定非 Tensor 参数类型

目前，to_static 装饰器中的 input_spec 参数仅接收 InputSpec 类型对象。若被装饰函数的参数列表除了 Tensor 类型，还包含其他如 Int、String 等非 Tensor 类型时，推荐在函数中使用 kwargs 形式定义非 Tensor 参数，如下述样例中的 use_act 参数。

```
class SimpleNet(Layer):
    def __init__(self, ):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)
        self.relu = paddle.nn.ReLU()

    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x')])
    def forward(self, x, use_act=False):
        out = self.linear(x)
        if use_act:
            out = self.relu(out)
        return out

net = SimpleNet()
adam = paddle.optimizer.Adam(parameters=net.parameters())

# train model
batch_num = 10
for step in range(batch_num):
    x = paddle.rand([4, 10], 'float32')
    use_act = (step%2 == 0)
    out = net(x, use_act)
    loss = paddle.mean(out)
    loss.backward()
    adam.minimize(loss)
    net.clear_gradients()

# save inference model with use_act=False
paddle.jit.save(net, model_path='./simple_net')
```

在上述样例中，step 为奇数时，use_act 取值为 False ； step 为偶数时，use_act 取值为 True 。动转静支持非 Tensor 参数在训练时取不同的值，且保证了取值不同的训练过程都可以更新模型的网络参数，行为与动态图一致。

kwargs 参数的默认值主要用于保存推理模型。在借助 paddle.jit.save 保存预测模型时，动转静会根据 input_spec 和 kwargs 的默认值保存推理模型和网络参数。因此建议将 kwargs 参数默认值设置为预测时的取值。

更多关于动转静 to_static 搭配 paddle.jit.save/load 的使用方式，可以参考模型存储与载入 。

4.5 报错信息处理

本节内容将介绍使用动态图转静态图（下文简称：动转静）功能发生异常时，ProgramTranslator的动转静报错模块对报错信息做的处理，以帮助您更好地理解动转静报错信息。使用动转静功能运行动态图代码时，内部可以分为 2 个步骤：动态图代码转换成静态图代码，运行静态图代码。接下来将分别介绍这 2 个步骤中的异常报错情况。

4.5.1 动转静过程中的异常

在动态图代码转换成静态图代码的过程中，如果 ProgramTranslator 无法转换一个函数时，将会显示警告信息，并尝试直接运行该函数。如下代码中，函数 `inner_func` 在调用前被转换成静态图代码，当 `x = inner_func(data)` 调用该函数时，不能重复转换，会给出警告信息：

```
import paddle
import numpy as np

@paddle.jit.to_static
def func():
    def inner_func(x):
        x_tensor = paddle.to_tensor(x)
        return x_tensor
    data = np.ones([3]).astype("int32")
    x = inner_func(data)
    return x
func()
```

ProgramTranslator 打印的警告信息如下：

```
2020-01-01 00:00:00,104-WARNING: <function inner_func at 0x125b3a550> doesn't have to␣
↪be transformed to static function because it has been transformed before, it will␣
↪be run as-is.
```

4.5.2 运行转换后的代码报错

如果在动转静后的静态图代码中发生异常，ProgramTranslator 会捕获该异常，增强异常报错信息，将静态图代码报错行映射到转换前的动态图代码，并重新抛出该异常。重新抛出的异常具有以下特点：

- 隐藏了部分对用户无用的动转静过程调用栈；
- 转换后的代码的异常信息，给出提示”In transformed code:”；
- 报错信息中包含了转换前的原始动态图代码，并给出提示”(* user code *)”；

例如，运行以下代码，在静态图构建时，即编译期会抛出异常：

```
import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    x = paddle.reshape(x, shape=[-1, -1])
    return x

func(np.ones([3, 2]))
```

运行结果：

```
Traceback (most recent call last):
  <ipython-input-13-f9c3ea702e3a> in <module>()
    func(np.ones([3, 2]))
  File "paddle/fluid/dygraph/dygraph_to_static/program_translator.py", line 352, in __
↪call__
    error_data.raise_new_exception()
  File "paddle/fluid/dygraph/dygraph_to_static/error.py", line 188, in raise_new_
↪exception
    raise new_exception
AssertionError: In transformed code:

    File "<ipython-input-13-f9c3ea702e3a>", line 7, in func (* user code *)
        x = paddle.reshape(x, shape=[-1, -1])
    File "paddle/fluid/layers/nn.py", line 6193, in reshape
        attrs["shape"] = get_attr_shape(shape)
    File "paddle/fluid/layers/nn.py", line 6169, in get_attr_shape
        "be -1. But received shape[%d] is also -1." % dim_idx)
    AssertionError: Only one dimension value of 'shape' in reshape can be -1. But␣
↪received shape[1] is also -1.
```

上述报错信息可以分为 3 点：

1. 报错栈中，涉及代码转换过程的信息栈默认会被隐藏，不进行展示，以减少干扰信息。
2. ProgramTranslator 处理后的报错信息中，会包含提示 “In transformed code:”，表示之后的报错信息栈，是在运行转换后的代码时的报错信息：

```
AssertionError: In transformed code:

    File "<ipython-input-13-f9c3ea702e3a>", line 7, in func (* user code *)
        x = paddle.reshape(x, shape=[-1, -1])
    File "paddle/fluid/layers/nn.py", line 6193, in reshape
```

(下页继续)

(续上页)

```
        attrs["shape"] = get_attr_shape(shape)
    File "paddle/fluid/layers/nn.py", line 6169, in get_attr_shape
        "be -1. But received shape[%d] is also -1." % dim_idx)
```

其中，File "<ipython-input-13-f9c3ea702e3a>", line 7, in func 是转换前的代码位置信息，x = paddle.reshape(x, shape=[-1, -1]) 是转换前用户的动态图代码。

3. 新的异常中，包含原始报错中的的报错信息，如下：

```
AssertionError: Only one dimension value of 'shape' in reshape can be -1. But␣
↪received shape[1] is also -1.
```

注解:

如果您想查看 Paddle 原生报错信息栈，即未被动转静模块处理过的报错信息栈，可以设置环境变量 TRANSLATOR_DISABLE_NEW_ERROR=1 关闭动转静报错模块。该环境变量默认值为 0，表示默认开启动转静报错模块。

运行以下代码，在静态图运行期会抛出异常：

```
@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    two = paddle.full(shape=[1], fill_value=2, dtype="int32")
    x = paddle.reshape(x, shape=[1, two])
    return x

func(np.ones([3]).astype("int32"))
```

运行结果：

```
Traceback (most recent call last):
  File "<ipython-input-57-c63d6a351262>", line 10, in <module>()
    func(np.ones([3]).astype("int32"))
  File "paddle/fluid/dygraph/dygraph_to_static/program_translator.py", line 352, in __
↪call__
    error_data.raise_new_exception()
  File "paddle/fluid/dygraph/dygraph_to_static/error.py", line 188, in raise_new_
↪exception
    raise new_exception
EnforceNotMet: In transformed code:

    File "<ipython-input-57-c63d6a351262>", line 7, in func
      x = paddle.reshape(x, shape=[1, two])
    File "paddle/tensor/manipulation.py", line 1347, in reshape
```

(下页继续)

(续上页)

```
      return paddle.fluid.layers.reshape(x=x, shape=shape, name=name)
    File "paddle/fluid/layers/nn.py", line 6209, in reshape
      "XShape": x_shape})
    File "paddle/fluid/layer_helper.py", line 43, in append_op
      return self.main_program.current_block().append_op(*args, **kwargs)
    File "paddle/fluid/framework.py", line 2880, in append_op
      attrs=kwargs.get("attrs", None))
    File "paddle/fluid/framework.py", line 1977, in __init__
      for frame in traceback.extract_stack():

    InvalidArgumentError: The 'shape' in ReshapeOp is invalid. The input tensor X
↪'size must be equal to the capacity of 'shape'. But received X's shape = [3], X's␣
↪size = 3, 'shape' is [1, 2], the capacity of 'shape' is 2.
      [Hint: Expected capacity == in_size, but received capacity:2 != in_size:3.] (at␣
↪/home/teamcity/work/ef54dc8a5b211854/paddle/fluid/operators/reshape_op.cc:222)
      [Hint: If you need C++ stacktraces for debugging, please set `FLAGS_call_stack_
↪level=2`.]
      [operator < reshape2 > error]  [operator < run_program > error]
```

上述异常中，除了隐藏部分报错栈、报错定位到转换前的动态图代码外，报错信息中隐藏了 C++ 报错栈，您可设置环境变量 `FLAGS_call_stack_level=2` 来展示 C++ 栈信息。

注解:

如果您想查看被隐藏的信息栈，可以设置环境变量 `TRANSLATOR_SIMPLIFY_NEW_ERROR=0`。该环境变量默认值为 1，表示隐藏冗余的报错信息栈。

4.6 调试方法

本节内容将介绍动态图转静态图（下文简称：动转静）推荐的几种调试方法。

注解:

请确保转换前的动态图代码能够成功运行，建议使用 paddle.jit.ProgramTranslator().enable(False)关闭动转静功能，直接运行动态图，如下：

```
import paddle
import numpy as np
# 关闭动转静动能
paddle.jit.ProgramTranslator().enable(False)

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
```

(下页继续)

(续上页)

```
    if x > 3:
        x = x - 1
    return x

func(np.ones([3, 2]))
```

4.6.1 断点调试

使用动转静功能时，您可以使用断点调试代码。例如，在代码中，调用 pdb.set_trace()：

```
import pdb

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    pdb.set_trace()
    if x > 3:
        x = x - 1
    return x
```

执行以下代码，将会在转化后的静态图代码中使用调试器：

```
func(np.ones([3, 2]))
```

运行结果：

```
> /tmp/tmpR809hf.py(6)func()
-> def true_fn_0(x):
(Pdb) n
> /tmp/tmpR809hf.py(6)func()
-> def false_fn_0(x):
...
```

如果您想在原始的动态图代码中使用调试器，请先调用 paddle.jit.ProgramTranslator().enable(False)，如下：

```
paddle.jit.ProgramTranslator().enable(False)
func(np.ones([3, 2]))
```

运行结果：

```
> <ipython-input-22-0bd4eab35cd5>(10)func()
-> if x > 3:
```

(下页继续)

(续上页)

```
...
```

4.6.2 打印转换后的代码

您可以打印转换后的静态图代码，有 2 种方法：

1. 使用被装饰后的函数的 code 属性

   如下代码中，装饰器 paddle.jit.to_static 会将函数 func 转化为一个类对象 StaticFunction，可以使用 StaticFunction 的 code 属性来获得转化后的代码。

```
@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if x > 3:
        x = x - 1
    return x

print(func.code)
```

   运行结果：

```
def func(x):
    x = paddle.nn.functional.assign(x)

    def true_fn_0(x):
        x = x - 1
        return x

    def false_fn_0(x):
        return x
    x = paddle.jit.dy2static.convert_ifelse(x > 3, true_fn_0, false_fn_0, (x,),␣
↪(x,), (x,))
    return x
```

2. 使用 set_code_level(level=100, also_to_stdout=False) 或环境变量 TRANSLATOR_CODE_LEVEL=level

   通过调用 set_code_level 或设置环境变量 TRANSLATOR_CODE_LEVEL，可以在日志中查看转换后的代码：

```
@paddle.jit.to_static
   def func(x):
   x = paddle.to_tensor(x)
```

(下页继续)

(续上页)

```
    if x > 3:
        x = x - 1
    return x

paddle.jit.set_code_level() # 也可设置 os.environ["TRANSLATOR_CODE_LEVEL"] = '100',
效果相同
func(np.ones([1]))
```

运行结果：

```
2020-XX-XX 00:00:00,980 Dynamic-to-Static INFO: After the level 100 ast␣
↪transformer: 'All Transformers', the transformed code:
def func(x):
    x = paddle.nn.functional.assign(x)

    def true_fn_0(x):
        x = x - 1
        return x

    def false_fn_0(x):
        return x
    x = paddle.jit.dy2static.convert_ifelse(x > 3, true_fn_0, false_fn_0, (x,),␣
↪(x,), (x,))
    return x
```

此外，如果您想将转化后的代码也输出到 sys.stdout，可以设置参数 also_to_stdout 为 True，否则将仅输出到 sys.stderr。set_code_level 函数可以设置查看不同的 AST Transformer 转化后的代码，详情请见 set_code_level。

4.6.3 使用 print

print 函数可以用来查看变量，该函数在动转静中会被转化。当仅打印 Paddle Tensor 时，实际运行时会被转换为 Paddle 算子 Print，否则仍然运行 print。

```
@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)

    # 打印 x, x 是 Paddle Tensor, 实际运行时会运行 Paddle Print(x)
    print(x)

    # 打印注释, 非 Paddle Tensor, 实际运行时仍运行 print
    print("Here call print function.")
```

(下页继续)

(续上页)

```
    if len(x) > 3:
        x = x - 1
    else:
        x = paddle.ones(shape=[1])
    return x

func(np.ones([1]))
```

运行结果：

```
Variable: assign_0.tmp_0
  - lod: {}
  - place: CPUPlace
  - shape: [1]
  - layout: NCHW
  - dtype: double
  - data: [1]
Here call print function.
```

4.6.4 日志打印

ProgramTranslator 在日志中记录了额外的调试信息，以帮助您了解动转静过程中函数是否被成功转换。您可以调用 `paddle.jit.set_verbosity(level=0, also_to_stdout=False)` 或设置环境变量 `TRANSLATOR_VERBOSITY=level` 来设置日志详细等级，并查看不同等级的日志信息。目前，`level` 可以取值 0-3：

- 0: 无日志
- 1: 包括了动转静转化流程的信息，如转换前的源码、转换的可调用对象
- 2: 包括以上信息，还包括更详细函数转化日志
- 3: 包括以上信息，以及更详细的动转静日志

  注意:

  日志中包括了源代码等信息，请在共享日志前确保它不包含敏感信息。

可以在代码运行前调用 `paddle.jit.set_verbosity` 控制日志详细程度：

```
paddle.jit.set_verbosity(3)
```

或者设置环境变量 `TRANSLATOR_VERBOSITY`：

```
import os
os.environ["TRANSLATOR_VERBOSITY"] = '3'
```

运行结果：

```
2020-XX-XX 00:00:00,123 Dynamic-to-Static INFO: (Level 1) Source code:
@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if len(x) > 3:
        x = x - 1
    else:
        x = paddle.ones(shape=[1])
    return x

2020-XX-XX 00:00:00,152 Dynamic-to-Static INFO: (Level 1) Convert callable object:␣
↪convert <built-in function len>.
```

此外，如果您想将日志也输出到 `sys.stdout`，可以设置参数 `also_to_stdout` 为 True，否则将仅输出到 `sys.stderr`，详情请见 set_verbosity。

CHAPTER 5

推理部署

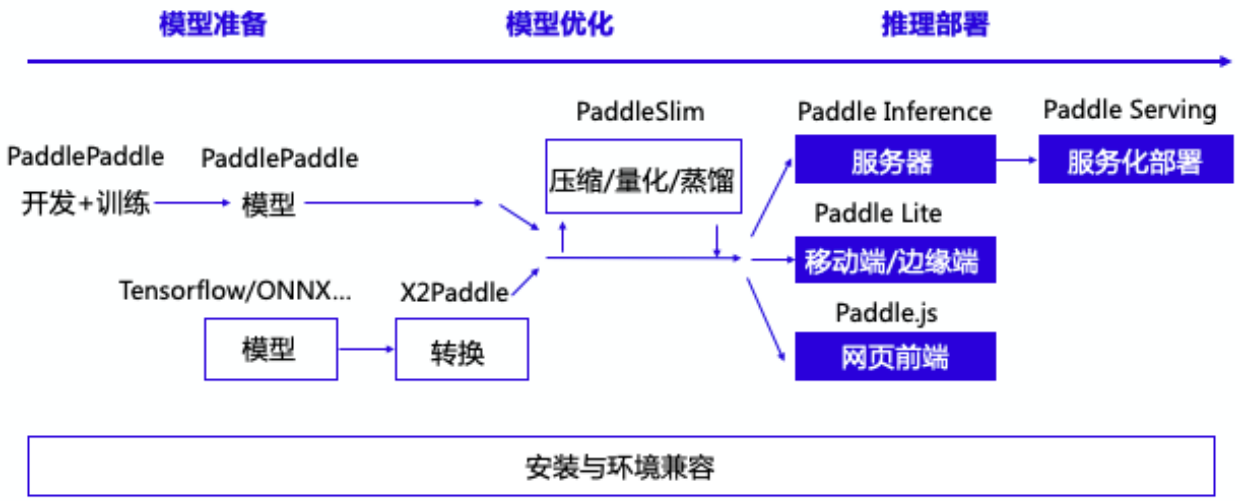
5.1 飞桨推理产品简介

作为飞桨生态重要的一部分，飞桨提供了多个推理产品，完整承接深度学习模型应用的最后一公里。

整体上分，推理产品主要包括如下子产品

名称	英文表示	适用场景
飞桨原生推理库	Paddle Inference	高性能服务器端、云端推理
飞桨服务化推理框架	Paddle Serving	自动服务、模型管理等高阶功能
飞桨轻量化推理引擎	Paddle Lite	移动端、物联网等
飞桨前端推理引擎	Paddle.js	浏览器中推理、小程序等

各产品在推理生态中的关系如下



用户使用飞桨推理产品的工作流如下

1. 获取一个飞桨的推理模型，其中有两种方法
    1. 利用飞桨训练得到一个推理模型
    2. 用 X2Paddle 工具从第三方框架（比如 TensorFlow 或者 Caffe 等）产出的模型转化
2. （可选）对模型进行进一步优化，PaddleSlim 工具可以对模型进行压缩，量化，裁剪等工作，显著提升模型执行的速度性能，降低资源消耗
3. 将模型部署到具体的推理产品上

5.1.1 服务器端部署

PaddlePaddle 提供了 C++，C 和 Python 的 API 来支持模型的部署上线。

安装与编译 Linux 预测库

直接下载安装

版本说明	预测库 (1.8.5 版本)	预测库 (2.0.0 版本)	预测库 (develop 版本)
ubuntu14.04_cpu_avx_mkl_gcc482	fluid_inference.tgz	paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cpu_avx_openblas_gcc482	fluid_inference.tgz	paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cpu_noavx_openblas_gcc482	fluid_inference.tgz	paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cuda9.0_cudnn7_avx_mkl_gcc482	fluid_inference.tgz	paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cuda9.0_cudnn7_avx_openblas_gcc482		paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cuda10.0_cudnn7_avx_mkl_gcc482	fluid_inference.tgz	paddle_inference.tgz	paddle_inference.tgz
ubuntu14.04_cuda10.1_cudnn7.6_avx_mkl_trt6_gcc482	fluid_inference.tgz		
ubuntu14.04_cuda10.1_cudnn7.6_avx_mkl_trt6_gcc82		paddle_inference.tgz	
ubuntu14.04_cuda10.2_cudnn8_avx_mkl_trt7_gcc82		paddle_inference.tgz	
ubuntu14.04_cuda11_cudnn8_avx_mkl_trt7_gcc82		paddle_inference.tgz	
nv_jetson_cuda10_cudnn7.6_trt6_all(jetpack4.3, for all Jetson devices)		paddle_inference.tar.gz	
nv_jetson_cuda10_cudnn7.6_trt6_nano(jetpack4.3, for Nano)		paddle_inference.tar.gz	
nv_jetson_cuda10_cudnn7.6_trt6_tx2(jetpack4.3, for TX2 series)		paddle_inference.tar.gz	
nv_jetson_cuda10_cudnn7.6_trt6_xavier(jetpack4.3, for AGX Xavier and Xavier NX)		paddle_inference.tar.gz	

从源码编译

用户也可以从 PaddlePaddle 核心代码编译 C++ 预测库，只需在编译时配制下面这些编译选项：

选项	值	说明
CMAKE_BUILD_TYPE	Release	编译方式，仅使用预测库设为 Release 即可
FLUID_INFERENCE_INSTALL_DIR	安装路径	预测库安装路径
WITH_PYTHON	OFF(推荐)	编译 python 预测库与 whl 包
ON_INFER	ON(推荐)	预测时使用，必须设为 ON
WITH_GPU	ON/OFF	编译支持 GPU 的预测库
WITH_MKL	ON/OFF	编译支持 MKL 的预测库
WITH_MKLDNN	ON/OFF	编译支持 MKLDNN 的预测库
WITH_XBYAK	ON	使用 XBYAK 编译，在 jetson 硬件上编译需要设置为 OFF
WITH_TENSORRT	OFF	编译支持 NVIDIA TensorRT 的预测库，需要另外配置 TENSORRT_ROOT 选项指定 TRT 根目录

建议按照推荐值设置，以避免链接不必要的库。其它可选编译选项按需进行设定。

首先从 github 拉取最新代码

```
git clone https://github.com/paddlepaddle/Paddle
cd Paddle
# 建议使用 git checkout 切换到 Paddle 稳定的版本，如:
git checkout release/2.0
```

note: 如果您是多卡机器，建议安装 NCCL；如果您是单卡机器则可以在编译时显示指定 WITH_NCCL=OFF 来跳过这一步。注意如果 WITH_NCCL=ON，且没有安装 NCCL，则编译会报错。

```
git clone https://github.com/NVIDIA/nccl.git
cd nccl
make -j4
make install
```

Server 端预测库源码编译

下面的代码片段配制编译选项并进行编译（需要将 PADDLE_ROOT 替换为 PaddlePaddle 预测库的安装路径，WITH_NCCL 根据实际情况进行修改）：

```
PADDLE_ROOT=/path/of/paddle
cd Paddle
mkdir build
cd build
cmake -DFLUID_INFERENCE_INSTALL_DIR=$PADDLE_ROOT \
      -DCMAKE_BUILD_TYPE=Release \
```

(下页继续)

(续上页)

```
      -DWITH_PYTHON=OFF \
      -DWITH_MKL=OFF \
      -DWITH_GPU=OFF  \
      -DON_INFER=ON \
      -DWITH_NCCL=OFF \
      ..
make
make inference_lib_dist
```

NVIDIA Jetson 嵌入式硬件预测库源码编译

NVIDIA Jetson 是 NVIDIA 推出的嵌入式 AI 平台，Paddle Inference 支持在 NVIDIA Jetson 平台上编译预测库。具体步骤如下：

1. 准备环境

   开启硬件性能模式

   ```
   sudo nvpmodel -m 0 && sudo jetson_clocks
   ```

   如果硬件为 Nano，增加 swap 空间

   ```
   # 增加 DDR 可用空间，Xavier 默认内存为 16G，所以内存足够，如想在 Nano 上尝试，
   请执行如下操作。
   sudo fallocate -l 5G /var/swapfile
   sudo chmod 600 /var/swapfile
   sudo mkswap /var/swapfile
   sudo swapon /var/swapfile
   sudo bash -c 'echo "/var/swapfile swap swap defaults 0 0" >> /etc/
   ↪fstab'
   ```

2. 编译 Paddle Inference 预测库

   ```
   cd Paddle
   mkdir build
   cd build
   cmake .. \
     -DWITH_CONTRIB=OFF \
     -DWITH_MKL=OFF  \
     -DWITH_MKLDNN=OFF \
     -DWITH_TESTING=OFF \
     -DCMAKE_BUILD_TYPE=Release \
     -DON_INFER=ON \
     -DWITH_PYTHON=OFF \
     -DWITH_XBYAK=OFF  \
   ```

(下页继续)

(续上页)

```
  -DWITH_NV_JETSON=ON
make -j4
# 生成预测 lib
make inference_lib_dist -j4
```

3. 样例测试

   请参照官网样例：https://www.paddlepaddle.org.cn/documentation/docs/zh/advanced_guide/performance_improving/inference_improving/paddle_tensorrt_infer.html#id2

FAQ

1. 报错：

   ```
   ERROR: ../aarch64-linux-gpn/crtn.o: Too many open files.
   ```

   则增加系统同一时间最多可开启的文件数至 2048

   ```
   ulimit -n 2048
   ```

2. 编译卡住

   可能是下载第三方库较慢的原因，耐心等待或 kill 掉编译进程重新编译

3. 使用 TensorRT 报错 IPluginFactory 或 IGpuAllocator 缺少虚析构函数

   下载安装 TensorRT 后，在 NvInfer.h 文件中为 class IPluginFactory 和 class IGpuAllocator 分别添加虚析构函数：

   ```
   virtual ~IPluginFactory() {};
   virtual ~IGpuAllocator() {};
   ```

成功编译后，使用 C++ 预测库所需的依赖（包括：（1）编译出的 PaddlePaddle 预测库和头文件；（2）第三方链接库和头文件；（3）版本信息与编译选项信息）均会存放于 PADDLE_ROOT 目录中。目录结构如下：

```
PaddleRoot/
├── CMakeCache.txt
├── paddle
│   ├── include
│   │   ├── paddle_anakin_config.h
│   │   ├── paddle_analysis_config.h
│   │   ├── paddle_api.h
│   │   ├── paddle_inference_api.h
│   │   ├── paddle_mkldnn_quantizer_config.h
│   │   └── paddle_pass_builder.h
│   └── lib
│       ├── libpaddle_fluid.a
```

(下页继续)

(续上页)

```
|       └── libpaddle_fluid.so
├── third_party
|   └── install
|       ├── gflags
|       ├── glog
|       ├── mkldnn
|       ├── mklml
|       └── protobuf
└── version.txt
```

version.txt 中记录了该预测库的版本信息，包括 Git Commit ID、使用 OpenBlas 或 MKL 数学库、CUDA/CUDNN 版本号，如：

```
GIT COMMIT ID: 0231f58e592ad9f673ac1832d8c495c8ed65d24f
WITH_MKL: ON
WITH_MKLDNN: ON
WITH_GPU: ON
CUDA version: 10.1
CUDNN version: v7
```

安装与编译 Windows 预测库

直接下载安装

硬件环境

测试环境硬件配置：

从源码编译

用户也可以从 PaddlePaddle 核心代码编译 C++ 预测库，只需在编译时配制下面这些编译选项：

请按照推荐值设置，以避免链接不必要的库。其它可选编译选项按需进行设定。

更多具体编译选项含义请参见编译选项表

Windows 下安装与编译预测库步骤：(在 Windows 命令提示符下执行以下指令)

1. 将 PaddlePaddle 的源码 clone 在当下目录的 Paddle 文件夹中，并进入 Paddle 目录，创建 build 目录：

```
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle
# 创建并进入 build 目录
```

(下页继续)

(续上页)

```
mkdir build
cd build
```

2. 执行 cmake：

   - 编译 CPU 预测库

```
cmake .. -G "Visual Studio 14 2015" -A x64 -T host=x64 -DCMAKE_BUILD_TYPE=Release␣
↪-DWITH_MKL=ON -DWITH_GPU=OFF -DON_INFER=ON -DWITH_PYTHON=OFF

# Windows 默认使用 /MT 模式进行编译，如果想使用 /MD 模式，请使用以下命令。如不清楚两者的区别，
请使用上面的命令
cmake .. -G "Visual Studio 14 2015" -A x64 -T host=x64 -DCMAKE_BUILD_TYPE=Release␣
↪-DWITH_MKL=ON -DWITH_GPU=OFF -DON_INFER=ON -DWITH_PYTHON=OFF -DMSVC_STATIC_
↪CRT=OFF
```

   - 编译 GPU 预测库:

```
# -DCUDA_TOOLKIT_ROOT_DIR 为你所安装的 cuda 根目录，例如-DCUDA_TOOLKIT_ROOT_DIR="C:/
↪Program Files/NVIDIA GPU Computing Toolkit/CUDA/v10.0"
cmake .. -G "Visual Studio 14 2015" -A x64 -T host=x64 -DCMAKE_BUILD_TYPE=Release␣
↪-DWITH_MKL=ON -DWITH_GPU=ON -DON_INFER=ON -DWITH_PYTHON=OFF -DCUDA_TOOLKIT_ROOT_
↪DIR=YOUR_CUDA_PATH
```

3. 使用 Blend for Visual Studio 2015 打开 `paddle.sln` 文件，选择平台为 `x64`，配置为 `Release`，编译 inference_lib_dist 项目。操作方法：在 Visual Studio 中选择相应模块，右键选择”生成”（或者”build”）

编译成功后，使用 C++ 预测库所需的依赖（包括：1. 编译出的 PaddlePaddle 预测库和头文件；2. 第三方链接库和头文件；3. 版本信息与编译选项信息）均会存放于 `fluid_inference_install_dir` 目录中。

version.txt 中记录了该预测库的版本信息，包括 Git Commit ID、使用 OpenBlas 或 MKL 数学库、CUDA/CUDNN 版本号、C++ 编译器版本，如：

```
GIT COMMIT ID: 264e76cae6861ad9b1d4bcd8c3212f7a78c01e4d
WITH_MKL: ON
WITH_MKLDNN: ON
WITH_GPU: ON
CUDA version: 10.0
CUDNN version: v7.4
CXX compiler version: 19.0.24215.1
```

编译预测 demo

硬件环境

测试环境硬件配置：

软件要求

请您严格按照以下步骤进行安装，否则可能会导致安装失败！

安装 Visual Studio 2015 update3

安装 Visual Studio 2015，安装选项中选择安装内容时勾选自定义，选择安装全部关于 c，c++，vc++ 的功能。

其他要求

1. 你需要直接下载 Windows 预测库或者从 Paddle 源码编译预测库，确保 windows 预测库存在。
2. 你需要下载 Paddle 源码，确保 demo 文件和脚本文件存在：

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

编译 demo

Windows 下编译预测 demo 步骤：(在 Windows 命令提示符下执行以下指令)

使用脚本编译运行

进入到 demo_ci 目录，运行脚本 `run_windows_demo.bat`，根据提示按需输入参数:

```
# path 为下载 Paddle 的目录
cd path\Paddle\paddle\fluid\inference\api\demo_ci
run_windows_demo.bat
```

其中，run_windows_demo.bat 的部分选项如下：

```
gpu_inference=Y # 是否使用 GPU 预测库，默认使用 CPU 预测库
use_mkl=Y # 该预测库是否使用 MKL，默认为 Y
use_gpu=Y  # 是否使用 GPU 进行预测，默认为 N。使用 GPU 预测需要下载 GPU 版本预测库

paddle_inference_lib=path\fluid_inference_install_dir # 设置 paddle 预测库的路径
cuda_lib_dir=path\lib\x64  # 设置 cuda 库的路径
vcvarsall_dir=path\vc\vcvarsall.bat  # 设置 visual studio # 本机工具命令提示符路径
```

手动编译运行

1. 进入 demo_ci 目录，创建并进入 build 目录

```
# path 为下载 Paddle 的目录
cd path\Paddle\paddle\fluid\inference\api\demo_ci
mkdir build
cd build
```

2. 执行 cmake（cmake 可以在官网进行下载，并添加到环境变量中):

   - 使用 CPU 预测库编译 demo

```
# -DDEMO_NAME 是要编译的文件
# -DDPADDLE_LIB 是预测库目录，例如-DPADDLE_LIB=D:\fluid_inference_install_dir
cmake .. -G "Visual Studio 14 2015" -A x64 -T host=x64 -DWITH_GPU=OFF -DWITH_
↪MKL=ON -DWITH_STATIC_LIB=ON ^
-DCMAKE_BUILD_TYPE=Release -DDEMO_NAME=simple_on_word2vec -DPADDLE_LIB=path_to_
↪the_paddle_lib -DMSVC_STATIC_CRT=ON
```

   - 使用 GPU 预测库编译 demo

```
# -DCUDA_LIB CUDA 的库目录，例如-DCUDA_LIB=D:\cuda\lib\x64
cmake .. -G "Visual Studio 14 2015" -A x64 -T host=x64 -DWITH_GPU=ON -DWITH_
↪MKL=ON -DWITH_STATIC_LIB=ON ^
-DCMAKE_BUILD_TYPE=Release -DDEMO_NAME=simple_on_word2vec -DPADDLE_LIB=path_to_
↪the_paddle_lib -DMSVC_STATIC_CRT=ON -DCUDA_LIB=YOUR_CUDA_LIB
```

3. 使用 Blend for Visual Studio 2015 打开 `cpp_inference_demo.sln` 文件，选择平台为 `x64`，配置为 `Release`，编译 simple_on_word2vec 项目。操作方法: 在 Visual Studio 中选择相应模块，右键选择”生成”（或者”build”）

4. 下载模型并解压到当前目录，执行命令:

```
# 开启 GLOG
set GLOG_v=100
# 进行预测，path 为模型解压后的目录
Release\simple_on_word2vec.exe --dirname=path\word2vec.inference.model
```

实现一个简单预测 demo

完整的代码示例

本示例使用了 AnalysisConfig 管理 AnalysisPredictor 的预测配置，提供了模型路径设置、预测引擎运行设备选择以及使用 ZeroCopyTensor 管理输入/输出的设置。具体步骤如下：

1. 创建 AnalysisConfig

```
AnalysisConfig config;
config->SwitchUseFeedFetchOps(false);  // 关闭 feed 和 fetch OP 使用，使用 ZeroCopy 接口必须设置此项
// config->EnableUseGpu(100 /* 设定 GPU 初始显存池为 MB*/,  0 /* 设定 GPU ID 为 0*/);//开启 GPU 预测
```

2. 在 config 中设置模型和参数路径

   从磁盘加载模型时，根据模型和参数文件存储方式不同，设置 AnalysisConfig 加载模型和参数的路径有两种形式，此处使用 combined 形式：

   - 非 combined 形式：模型文件夹 `model_dir` 下存在一个模型文件和多个参数文件时，传入模型文件夹路径，模型文件名默认为 `__model__`。

```
config->SetModel("path\\model_dir\\__model__")
```

   - combined 形式：模型文件夹 `model_dir` 下只有一个模型文件 `__model__` 和一个参数文件 `__params__` 时，传入模型文件和参数文件路径。

```
config->SetModel("path\\model_dir\\__model__", "path\\model_dir\\__params__");
```

3. 创建 predictor，准备输入数据

```
std::unique_ptr<PaddlePredictor> predictor = CreatePaddlePredictor(config);
int batch_size = 1;
int channels = 3; // channels, height, width 三个参数必须与模型中对应输入的 shape 一致
int height = 300;
int width = 300;
int nums = batch_size * channels * height * width;

float* input = new float[nums];
for (int i = 0; i < nums; ++i) input[i] = 0;
```

4. 使用 ZeroCopyTensor 管理输入

```
// 通过创建的 AnalysisPredictor 获取输入 Tensor，该 Tensor 为 ZeroCopyTensor
auto input_names = predictor->GetInputNames();
auto input_t = predictor->GetInputTensor(input_names[0]);
```

(下页继续)

(续上页)

```
// 对 Tensor 进行 reshape，将准备好的输入数据从 CPU 拷贝到 ZeroCopyTensor 中
input_t->Reshape({batch_size, channels, height, width});
input_t->copy_from_cpu(input);
```

5. 运行预测引擎

```
predictor->ZeroCopyRun();
```

6. 使用 ZeroCopyTensor 管理输出

```
auto output_names = predictor->GetOutputNames();
auto output_t = predictor->GetOutputTensor(output_names[0]);
std::vector<int> output_shape = output_t->shape();
int out_num = std::accumulate(output_shape.begin(), output_shape.end(), 1,
                              std::multiplies<int>());

out_data.resize(out_num);
output_t->copy_to_cpu(out_data.data()); // 将 ZeroCopyTensor 中数据拷贝到 cpu 中，得到输出数据
delete[] input;
```

Note: 关于 AnalysisPredictor 的更多介绍，请参考C++ 预测 API 介绍

C++ 预测 API 介绍

为了更简单方便地预测部署，PaddlePaddle 提供了一套高层 C++ API 预测接口。下面是详细介绍。

如果您在使用 2.0 之前的 Paddle，请参考旧版 API文档，升级到新版 API 请参考推理升级指南。

内容

- 使用 *Predictor* 进行高性能预测
- 使用 *Config* 管理预测配置
- 使用 *Tensor* 管理输入/输出
- 使用 *PredictorPool* 在多线程下进行预测
- *C++* 预测样例编译测试
- 性能调优
- 推理升级指南
- *C++ API*

使用 Predictor 进行高性能预测

Paddle Inference 采用 Predictor 进行预测。Predictor 是一个高性能预测引擎，该引擎通过对计算图的分析，完成对计算图的一系列的优化（如 OP 的融合、内存/显存的优化、MKLDNN，TensorRT 等底层加速库的支持等），能够大大提升预测性能。

为了展示完整的预测流程，下面是一个使用 Predictor 进行预测的完整示例，其中涉及到的具体概念和配置会在后续部分展开详细介绍。

Predictor 预测示例

```
#include "paddle_inference_api.h"

namespace paddle_infer {
void CreateConfig(Config* config, const std::string& model_dirname) {
  // 模型从磁盘进行加载
  config->SetModel(model_dirname + "/model",
                   model_dirname + "/params");
  // config->SetModel(model_dirname);
  // 如果模型从内存中加载，可以使用 SetModelBuffer 接口
  // config->SetModelBuffer(prog_buffer, prog_size, params_buffer, params_size);
  config->EnableUseGpu(100 /* 设定 GPU 初始显存池为 MB*/,  0 /* 设定 GPU ID 为 0*/); //开启
GPU 预测

  /* for cpu
  config->DisableGpu();
  config->EnableMKLDNN();   // 开启 MKLDNN 加速
  config->SetCpuMathLibraryNumThreads(10);
  */

  config->SwitchIrDebug(true);          // 可视化调试选项，若开启，则会在每个图优化过程后生成
dot 文件
  // config->SwitchIrOptim(false);      // 默认为 true。如果设置为 false，关闭所有优化
  // config->EnableMemoryOptim();      // 开启内存/显存复用
}

void RunAnalysis(int batch_size, std::string model_dirname) {
  // 1. 创建 Config
  Config config;
  CreateConfig(&config, model_dirname);

  // 2. 根据 config 创建 predictor，并准备输入数据，此处以全 0 数据为例
  auto predictor = CreatePredictor(config);
  int channels = 3;
```

(下页继续)

(续上页)

```
  int height = 224;
  int width = 224;
  float input[batch_size * channels * height * width] = {0};

  // 3. 创建输入
  // 使用了 ZeroCopy 接口，可以避免预测中多余的 CPU copy，提升预测性能
  auto input_names = predictor->GetInputNames();
  auto input_t = predictor->GetInputHandle(input_names[0]);
  input_t->Reshape({batch_size, channels, height, width});
  input_t->CopyFromCpu(input);

  // 4. 运行预测引擎
  CHECK(predictor->Run());

  // 5. 获取输出
  std::vector<float> out_data;
  auto output_names = predictor->GetOutputNames();
  auto output_t = predictor->GetOutputHandle(output_names[0]);
  std::vector<int> output_shape = output_t->shape();
  int out_num = std::accumulate(output_shape.begin(), output_shape.end(), 1,␣
↪std::multiplies<int>());

  out_data.resize(out_num);
  output_t->CopyToCpu(out_data.data());
}
}  // namespace paddle_infer

int main() {
  // 模型下载地址 http://paddle-inference-dist.cdn.bcebos.com/tensorrt_test/mobilenet.
↪tar.gz
  paddle_infer::RunAnalysis(1, "./mobilenet");
  return 0;
}
```

使用 Config 管理预测配置

Config 管理 Predictor 的预测配置，提供了模型路径设置、预测引擎运行设备选择以及多种优化预测流程的选项。配置方法如下：

通用优化配置

```
config->SwitchIrOptim(true);  // 开启计算图分析优化，包括 OP 融合等
config->EnableMemoryOptim();  // 开启内存/显存复用
```

设置模型和参数路径

从磁盘加载模型时，根据模型和参数文件存储方式不同，设置 Config 加载模型和参数的路径有两种形式：

- 非 combined 形式：模型文件夹 `model_dir` 下存在一个模型文件和多个参数文件时，传入模型文件夹路径，模型文件名默认为 `__model__`。

```
config->SetModel("./model_dir");
```

- combined 形式：模型文件夹 `model_dir` 下只有一个模型文件 `model` 和一个参数文件 `params` 时，传入模型文件和参数文件路径。

```
config->SetModel("./model_dir/model", "./model_dir/params");
```

配置 CPU 预测

```
config->DisableGpu();           // 禁用 GPU
config->EnableMKLDNN();             // 开启 MKLDNN，可加速 CPU 预测
config->SetCpuMathLibraryNumThreads(10);        // 设置 CPU Math 库线程数，CPU 核心数支持情
况下可加速预测
```

note

如果在输入 shape 为变长时开启 MKLDNN 加速预测，需要通过 `SetMkldnnCacheCapacity` 接口设置 MKLDNN 缓存的不同输入 shape 的数目，否则可能会出现内存泄漏。使用方法如下：

```
config->SetMkldnnCacheCapacity(100); // 缓存 100 个不同的输入 shape
```

配置 GPU 预测

```
config->EnableUseGpu(100, 0); // 初始化 100M 显存，使用 GPU ID 为 0
config->GpuDeviceId();        // 返回正在使用的 GPU ID
// 开启 TensorRT 预测，可提升 GPU 预测性能，需要使用带 TensorRT 的预测库
config->EnableTensorRtEngine(1 << 20              /*workspace_size*/,
                             batch_size        /*max_batch_size*/,
                             3                 /*min_subgraph_size*/,
                             PrecisionType::kFloat32 /*precision*/,
                             false             /*use_static*/,
                             false             /*use_calib_mode*/);
```

使用 Tensor 管理输入/输出

Tensor 是 Predictor 的输入/输出数据结构。

```
// 通过创建的 Predictor 获取输入和输出的 tensor
auto input_names = predictor->GetInputNames();
auto input_t = predictor->GetInputHandle(input_names[0]);
auto output_names = predictor->GetOutputNames();
auto output_t = predictor->GetOutputHandle(output_names[0]);

// 对 tensor 进行 reshape
input_t->Reshape({batch_size, channels, height, width});

// 通过 CopyFromCpu 接口，将 cpu 数据输入；通过 CopyToCpu 接口，将输出数据 copy 到 cpu
input_t->CopyFromCpu<float>(input_data /* 数据指针 */);
output_t->CopyToCpu(out_data /* 数据指针 */);

// 设置 LOD
std::vector<std::vector<size_t>> lod_data = {{0}, {0}};
input_t->SetLoD(lod_data);

// 获取 Tensor 数据指针
float *input_d = input_t->mutable_data<float>(PaddlePlace::kGPU);  // CPU 下使用
PaddlePlace::kCPU
int output_size;
float *output_d = output_t->data<float>(PaddlePlace::kGPU, &output_size);
```

使用 PredictorPool 在多线程下进行预测

PredictorPool 对 Predictor 进行管理。PredictorPool 对 Predictor 进行了简单的封装，通过传入 config 和 thread 的数目来完成初始化，在每个线程中，根据自己的线程 id 直接从池中取出对应的 Predictor 来完成预测过程。

```
# 服务初始化时，完成 PredictorPool 的初始化
PredictorPool pool(config, thread_num);

# 根据线程 id 来获取 Predictor
auto predictor = pool.Retrive(thread_id);

# 使用 Predictor 进行预测
...
```

C++ 预测样例编译测试

1. 下载或编译 paddle 预测库，参考安装与编译 C++ 预测库。
2. 下载预测样例并解压，进入 sample/inference 目录下。

   inference 文件夹目录结构如下：

   ```
   inference
   ├── CMakeLists.txt
   ├── mobilenet_test.cc
   ├── thread_mobilenet_test.cc
   ├── mobilenetv1
   |   ├── model
   |   └── params
   ├── run.sh
   └── run_impl.sh
   ```

   - mobilenet_test.cc 为单线程预测的 C++ 源文件
   - thread_mobilenet_test.cc 为多线程预测的 C++ 源文件
   - mobilenetv1 为模型文件夹
   - run.sh 为预测运行脚本文件
3. 配置编译与运行脚本

   编译运行预测样例之前，需要根据运行环境配置编译与运行脚本 run.sh。run.sh 的选项与路径配置的部分如下：

```
# 设置是否开启 MKL、GPU、TensorRT，如果要使用 TensorRT，必须打开 GPU
WITH_MKL=ON
WITH_GPU=OFF
USE_TENSORRT=OFF

# 按照运行环境设置预测库路径、CUDA 库路径、CUDNN 库路径、TensorRT 路径、模型路径
LIB_DIR=YOUR_LIB_DIR
CUDA_LIB_DIR=YOUR_CUDA_LIB_DIR
CUDNN_LIB_DIR=YOUR_CUDNN_LIB_DIR
TENSORRT_ROOT_DIR=YOUR_TENSORRT_ROOT_DIR
MODEL_DIR=YOUR_MODEL_DIR
```

按照实际运行环境配置 `run.sh` 中的选项开关和所需 lib 路径。

4. 编译与运行样例

```
sh run.sh
```

性能调优

CPU 下预测

1. 在 CPU 型号允许的情况下，尽量使用带 AVX 和 MKL 的版本。
2. 可以尝试使用 Intel 的 MKLDNN 加速。
3. 在 CPU 可用核心数足够时，可以将设置 `config->SetCpuMathLibraryNumThreads(num);` 中的 num 值调高一些。

GPU 下预测

1. 可以尝试打开 TensorRT 子图加速引擎，通过计算图分析，Paddle 可以自动将计算图中部分子图融合，并调用 NVIDIA 的 TensorRT 来进行加速，详细内容可以参考 使用 Paddle-TensorRT 库预测。

多线程预测

Paddle Inference 支持通过在不同线程运行多个 Predictor 的方式来优化预测性能，支持 CPU 和 GPU 环境。

使用多线程预测的样例详见 *C++ 预测样例编译测试*中下载的预测样例中的 `thread_mobilenet_test.cc` 文件。可以将 `run.sh` 中 `mobilenet_test` 替换成 `thread_mobilenet_test` 再执行

```
sh run.sh
```

即可运行多线程预测样例。

推理升级指南

2.0 对 API 做了整理，简化了写法，以及去掉了历史上冗余的概念。

新的 API 为纯增，原有 API 保持不变，在后续版本会逐步删除。

重要变化：

- 命名空间从 `paddle` 变更为 `paddle_infer`
- `PaddleTensor`, `PaddleBuf` 等被废弃，`ZeroCopyTensor` 变为默认 Tensor 类型，并更名为 `Tensor`
- 新增 `PredictorPool` 工具类简化多线程 predictor 的创建，后续也会增加更多周边工具
- `CreatePredictor` (原 `CreatePaddlePredictor`) 的返回值由 `unique_ptr` 变为 `shared_ptr` 以避免 Clone 后析构顺序出错的问题

API 变更

使用新 C++ API 的流程与之前完全一致，只有命名变化

```
#include "paddle_infernce_api.h"
using namespace paddle_infer;

Config config;
config.SetModel("xxx_model_dir");

auto predictor = CreatePredictor(config);

// Get the handles for the inputs and outputs of the model
auto input0 = predictor->GetInputHandle("X");
auto output0 = predictor->GetOutputHandle("Out");

for (...) {
    // Assign data to input0
  MyServiceSetData(input0);

  predictor->Run();

  // get data from the output0 handle
  MyServiceGetData(output0);
}
```

C++ API

CreatePredictor

```
std::shared_ptr<Predictor> CreatePredictor(const Config& config);
```

CreatePredictor 用来根据 Config 构建预测引擎。

示例：

```
// 设置 Config
Config config;
config.SetModel(FLAGS_model_dir);

// 根据 Config 创建 Predictor
std::shared_ptr<Predictor> predictor = CreatePredictor(config);
```

参数：

- config(Config) - 用于构建 Predictor 的配置信息

返回：Predictor 智能指针

返回类型：std::shared_ptr<Predictor>

GetVersion()

```
std::string GetVersion();
```

打印 Paddle Inference 的版本信息。

参数：

- None

返回：版本信息

返回类型：std::string

PlaceType

```
enum class PaddlePlace { kUNK };
using PlaceType = paddle::PaddlePlace;
```

PlaceType 为目标设备硬件类型，用户可以根据应用场景选择硬件平台类型。

枚举变量 PlaceType 的所有可能取值包括：

{kUNK, kCPU, kGPU}

PrecisionType

```
enum class Precision { kFloat32 };
using PrecisionType = paddle::AnalysisConfig::Precision;
```

PrecisionType 设置模型的运行精度，默认值为 kFloat32(float32)。

枚举变量 PrecisionType 的所有可能取值包括：

{kFloat32, kInt8, kHalf}

DataType

```
enum class PaddleDType { FLOAT32 };
using DataType = paddle::PaddleDType;
```

DataType 为模型中 Tensor 的数据精度，默认值为 FLOAT32(float32)。

枚举变量 DataType 的所有可能取值包括：

{FLOAT32, INT64, INT32, UINT8}

GetNumBytesOfDataType

```
int GetNumBytesOfDataType(DataType dtype);
```

获取各个 DataType 对应的字节数。

参数：

- dtype - DataType 枚举

返回：字节数

返回类型：int

Predictor

```
class Predictor;
```

Predictor 是 Paddle Inference 的预测器，由 CreatePredictor 根据 Config 进行创建。用户可以根据 Predictor 提供的接口设置输入数据、执行模型预测、获取输出等.

示例：

```
using namespace paddle_infer;

Config config;
config.SetModel("xxx_model_dir");

auto predictor = CreatePredictor(config);

// Get the handles for the inputs and outputs of the model
auto input0 = predictor->GetInputHandle("X");
auto output0 = predictor->GetOutputHandle("Out");

for (...) {
    // Assign data to input0
  MyServiceSetData(input0);

  predictor->Run();

  // get data from the output0 handle
  MyServiceGetData(output0);
}
```

GetInputNames()

获取所有输入 Tensor 的名称。

参数：

- None

返回：所有输入 Tensor 的名称

返回类型：std::vector<std::string>

GetOutputNames()

获取所有输出 Tensor 的名称。

参数：

- None

返回：所有输出 Tensor 的名称

返回类型：std::vector<std::string>

GetInputHandle(const std::string& name)

根据名称获取输入 Tensor 的句柄。

参数：

- name - Tensor 的名称

返回：指向 Tensor 的指针

返回类型：std::unique_ptr<Tensor>

GetOutputHandle(const std::string& name)

根据名称获取输出 Tensor 的句柄。

参数：

- name - Tensor 的名称

返回：指向 Tensor 的指针

返回类型：std::unique_ptr<Tensor>

Run()

执行模型预测，需要在 *** 设置输入数据后 *** 调用。

参数：

- None

返回：None

返回类型：void

ClearIntermediateTensor()

释放临时 tensor，将其所占空间归还显/内存池。

参数：

- None

返回：None

返回类型：void

TryShrinkMemory()

释放临时 tensor，并检查显/内存池中是否有可以释放的 chunk，若有则释放 chunk，降低显/内存占用（显/内存池可认为是 list<chunk> 组成，如果 chunk 空闲，则可通过释放 chunk 来降低显/内存占用），demo 示例可参考Paddle-Inference-Demo。

参数：

- None

返回：None

返回类型：void

Clone()

根据该 Predictor，克隆一个新的 Predictor，两个 Predictor 之间共享权重。

参数：

- None

返回：新的 Predictor

返回类型：std::unique_ptr<Predictor>

Tensor

```
class Tensor;
```

Tensor 是 Paddle Inference 的数据组织形式，用于对底层数据进行封装并提供接口对数据进行操作，包括设置 Shape、数据、LoD 信息等。

注意：用户应使用 Predictor 的 GetInputHandle 和 GetOuputHandle 接口获取输入/输出的 Tensor。

示例：

```
// 通过创建的 Predictor 获取输入和输出的 tensor
auto input_names = predictor->GetInputNames();
auto input_t = predictor->GetInputHandle(input_names[0]);
auto output_names = predictor->GetOutputNames();
auto output_t = predictor->GetOutputHandle(output_names[0]);

// 对 tensor 进行 reshape
input_t->Reshape({batch_size, channels, height, width});

// 通过 CopyFromCpu 接口，将 cpu 数据输入；通过 CopyToCpu 接口，将输出数据 copy 到 cpu
input_t->CopyFromCpu<float>(input_data /* 数据指针 */);
output_t->CopyToCpu(out_data /* 数据指针 */);

// 设置 LOD
std::vector<std::vector<size_t>> lod_data = {{0}, {0}};
input_t->SetLoD(lod_data);

// 获取 Tensor 数据指针
float *input_d = input_t->mutable_data<float>(PlaceType::kGPU);  // CPU 下使用
PlaceType::kCPU
int output_size;
float *output_d = output_t->data<float>(PlaceType::kGPU, &output_size);
```

Reshape(shape)

设置 Tensor 的维度信息。

参数：

- shape(const std::vector<int>&) - 维度信息

返回：None

返回类型：void

shape()

获取 Tensor 的维度信息。

参数：

- None

返回：Tensor 的维度信息

返回类型：std::vector<int>

CopyFromCpu(data)

```
template <typename T>
void CopyFromCpu(const T* data);
```

从 cpu 获取数据，设置到 tensor 内部。

示例：

```
// float* data = ...;
auto in_tensor = predictor->GetInputHandle("in_name");
in_tensor->CopyFromCpu(data);
```

参数：

- data(const T*) - cpu 数据指针

返回：None

返回类型：void

CopyToCpu(data)

```
template <typename T>
void CopyToCpu(T* data);
```

示例：

```
std::vector<float> data(100);
auto out_tensor = predictor->GetOutputHandle("out_name");
out_tensor->CopyToCpu(data.data());
```

参数：

- data(T*) - cpu 数据指针

返回：None

返回类型：void

data(place, size)

```
template <typename T>
T* data(PlaceType* place, int* size) const;
```

获取 Tensor 的底层数据的常量指针，用于读取 Tensor 数据。

示例：

```
PlaceType place;
int size;
auto out_tensor = predictor->GetOutputHandle("out_name");
float* data = out_tensor->data<float>(&place, &size);
```

参数：

- `place(PlaceType*)` - 获取 tensor 的 PlaceType
- `size(int*)` - 获取 tensor 的 size

返回：数据指针

返回类型：`T*`

mutable_data(place)

```
template <typename T>
T* mutable_data(PlaceType place);
```

获取 Tensor 的底层数据的指针，用于设置 Tensor 数据。

```
auto in_tensor = predictor->GetInputHandle("in_name");
float* data = out_tensor->mutable_data<float>(PlaceType::kCPU);
data[0] = 1.;
```

参数：

- `place(PlaceType)` - 设备信息

返回：`Tensor` 底层数据指针

返回类型：`T*`

SetLoD(lod)

设置 Tensor 的 LoD 信息。

参数:

- `lod(const std::vector<std::vector<size_t>>)` - Tensor 的 LoD 信息

返回: `None`

返回类型: `void`

lod()

获取 Tensor 的 LoD 信息

参数:

- `None`

返回: `Tensor` 的 LoD 信息

返回类型: `std::vector<std::vector<size_t>>`

type()

tensor 的 DataType 信息。

参数:

- `None`

返回: `Tensor` 的 DataType 信息

返回类型: `DataType`

name()

tensor 对应的 name。

参数:

- `None`

返回: `Tensor` 对应的 name

返回类型: `std::string`

Config

```
class Config;
```

Config 用来配置构建 Predictor 的配置信息，如模型路径、是否开启 gpu 等等。

示例：

```
Config config;
config.SetModel(FLAGS_model_dir);
config.DisableGpu();
config->SwitchIrOptim(false);      // 默认为 true。如果设置为 false，关闭所有优化
config->EnableMemoryOptim();      // 开启内存/显存复用
```

SetModel(const std::string& model_dir)

设置模型文件路径，当需要从磁盘加载非 combine 模式时使用。

参数：

- model_dir - 模型文件夹路径

返回：None

返回类型：void

model_dir()

获取模型文件夹路径。

参数：

- None

返回：模型文件夹路径

返回类型：string

SetModel(const std::string& prog, const std::string& params)

设置模型文件路径，当需要从磁盘加载 combine 模式时使用。

参数：

- prog - 模型文件路径
- params - 模型参数文件路径

返回：None

返回类型：void

SetProgFile(const std::string& prog)

设置模型文件路径。

参数：

- prog - 模型文件路径

返回：None

返回类型：void

prog_file()

获取模型文件路径。

参数：

- None

返回：模型文件路径

返回类型：string

SetParamsFile(const std::string& params)

设置模型参数文件路径。

参数：

- params - 模型文件路径

返回：None

返回类型：void

params_file()

获取模型参数文件路径。

参数：

- None

返回：模型参数文件路径

返回类型：string

SetModelBuffer(const char* prog_buffer, size_t prog_buffer_size, const char* params_buffer, size_t params_buffer_size)

从内存加载模型。

参数:

- prog_buffer - 内存中模型结构数据
- prog_buffer_size - 内存中模型结构数据的大小
- params_buffer - 内存中模型参数数据
- params_buffer_size - 内存中模型参数数据的大小

返回: None

返回类型: void

model_from_memory()

判断是否从内存中加载模型。

参数:

- None

返回: 是否从内存中加载模型

返回类型: bool

SetOptimCacheDir(const std::string& opt_cache_dir)

设置缓存路径。

参数:

- opt_cache_dir - 缓存路径

返回: None

返回类型: void

DisableFCPadding ()

关闭 fc padding。

参数:

- None

返回: None

返回类型: void

use_fc_padding()

判断是否启用 fc padding。

参数:

- None

返回: 是否启用 fc padding

返回类型: bool

EnableUseGpu(uint64_t memory_pool_init_size_mb, int device_id = 0)

启用 gpu。

参数:

- memory_pool_init_size_mb - 初始化分配的 gpu 显存，以 MB 为单位
- device_id - 设备 id

返回: None

返回类型: void

DisableGpu()

禁用 gpu。

参数:

- None

返回: None

返回类型: void

use_gpu()

是否启用 gpu。

参数:

- None

返回：是否启用 gpu

返回类型：bool

gpu_device_id()

获取 gpu 的 device id。

参数:

- None

返回：gpu 的 device id

返回类型：int

memory_pool_init_size_mb()

获取 gpu 的初始显存大小。

参数:

- None

返回：初始的显存大小

返回类型：int

fraction_of_gpu_memory_for_pool()

初始化显存占总显存的百分比

参数:

- None

返回：初始的显存占总显存的百分比

返回类型：float

EnableCUDNN()

启用 cudnn。

参数:

- None

返回：None

返回类型：void

cudnn_enabled()

是否启用 cudnn。

参数:

- None

返回：是否启用 cudnn

返回类型：bool

EnableXpu(int l3_workspace_size)

启用 xpu。

参数:

- l3_workspace_size - l3 cache 分配的显存大小

返回：None

返回类型：void

SwitchIrOptim(int x=true)

设置是否开启 ir 优化。

参数:

- x - 是否开启 ir 优化，默认打开

返回：None

返回类型：void

ir_optim()

是否开启 ir 优化。

参数:

- None

返回：是否开启 ir 优化

返回类型：bool

SwitchUseFeedFetchOps(int x = true)

设置是否使用 feed，fetch op，仅内部使用。

参数:

- x - 是否使用 feed, fetch op

返回：None

返回类型：void

use_feed_fetch_ops_enabled()

是否使用 feed，fetch op。

参数:

- None

返回：是否使用 feed，fetch op

返回类型：bool

SwitchSpecifyInputNames(bool x = true)

设置是否需要指定输入 tensor 的 name。

参数:

- x - 是否指定输入 tensor 的 name

返回：None

返回类型：void

specify_input_name()

是否需要指定输入 tensor 的 name。

参数：

- None

返回：是否需要指定输入 tensor 的 name

返回类型：bool

EnableTensorRtEngine(int workspace_size = 1 « 20, int max_batch_size = 1, int min_subgraph_size = 3, Precision precision = Precision::kFloat32, bool use_static = false, bool use_calib_mode = true)

设置是否启用 TensorRT。

参数：

- workspace_size - 指定 TensorRT 使用的工作空间大小
- max_batch_size - 设置最大的 batch 大小，运行时 batch 大小不得超过此限定值
- min_subgraph_size - Paddle-TRT 是以子图的形式运行，为了避免性能损失，当子图内部节点个数大于 min_subgraph_size 的时候，才会使用 Paddle-TRT 运行
- precision - 指定使用 TRT 的精度，支持 FP32（kFloat32），FP16（kHalf），Int8（kInt8）
- use_static - 如果指定为 true，在初次运行程序的时候会将 TRT 的优化信息进行序列化到磁盘上，下次运行时直接加载优化的序列化信息而不需要重新生成
- use_calib_mode - 若要运行 Paddle-TRT int8 离线量化校准，需要将此选项设置为 true

返回：None

返回类型：void

tensorrt_engine_enabled()

是否启用 tensorRT。

参数：

- None

返回：是否启用 tensorRT

返回类型：bool

SetTRTDynamicShapeInfo(std::map<std::string, std::vector> min_input_shape, std::map<std::string, std::vector> max_input_shape, std::map<std::string, std::vector> optim_input_shape, bool disable_trt_plugin_fp16 = false)

设置 tensorRT 的动态 shape。

参数:

- `min_input_shape` - tensorRT 子图支持动态 shape 的最小 shape
- `max_input_shape` - tensorRT 子图支持动态 shape 的最大 shape
- `optim_input_shape` - tensorRT 子图支持动态 shape 的最优 shape
- `disable_trt_plugin_fp16` - 设置 tensorRT 的 plugin 不在 fp16 精度下运行

返回：`None`

返回类型：`void`

EnableLiteEngine(PrecisionType precision_mode = Precsion::kFloat32, bool zero_copy = false, const std::vectorstd::string& passes_filter = {}, const std::vectorstd::string& ops_filter = {})

启用 lite 子图。

参数:

- `precision_mode` - lite 子图的运行精度
- `zero_copy` - 启用 zero_copy，lite 子图与 paddle inference 之间共享数据
- `passes_filter` - 设置 lite 子图的 pass
- `ops_filter` - 设置不使用 lite 子图运行的 op

返回：`None`

返回类型：`void`

lite_engine_enabled()

是否启用 lite 子图。

参数:

- `None`

返回：是否启用 lite 子图

返回类型：`bool`

SwitchIrDebug(int x = true)

设置是否在图分析阶段打印 ir，启用后会在每一个 pass 后生成 dot 文件。

参数：

- x - 是否打印 ir

返回：None

返回类型：void

EnableMKLDNN()

启用 mkldnn。

参数：

- None

返回：None

返回类型：void

SetMkldnnCacheCapacity(int capacity)

设置 mkldnn 针对不同输入 shape 的 cache 容量大小，MKLDNN cache 设计文档请参考链接

参数：

- capacity - cache 容量大小

返回：None

返回类型：void

mkldnn_enabled()

是否启用 mkldnn。

参数：

- None

返回：是否启用 mkldnn

返回类型：bool

SetMKLDNNOp(std::unordered_setstd::string op_list)

指定优先使用 mkldnn 加速的 op 列表。

参数:

- `op_list` - 优先使用 mkldnn 的 op 列表

返回: `None`

返回类型: `void`

EnableMkldnnQuantizer()

启用 mkldnn 量化。

参数:

- `None`

返回: `None`

返回类型: `void`

mkldnn_quantizer_enabled()

是否启用 mkldnn 量化。

参数:

- `None`

返回: 是否启用 mkldnn 量化

返回类型: `bool`

EnableMkldnnBfloat16()

启用 mkldnn bf16。

参数:

- `None`

返回: `None`

返回类型: `void`

mkldnn_bfloat16_enabled()

是否启用 mkldnn bf16。

参数：

- None

返回：是否启用 mkldnn bf16

返回类型：bool

mkldnn_quantizer_config()

返回 mkldnn 量化 config。

参数：

- None

返回：mkldnn 量化 config

返回类型：MkldnnQuantizerConfig

SetCpuMathLibraryNumThreads(int cpu_math_library_num_threads)

设置 cpu blas 库计算线程数。

参数：

- cpu_math_library_num_threads - blas 库计算线程数

返回：None

返回类型：void

cpu_math_library_num_threads()

cpu blas 库计算线程数。

参数：

- None

返回：cpu blas 库计算线程数。

返回类型：int

ToNativeConfig()

转化为 NativeConfig，不推荐使用。

参数：

- None

返回：当前 Config 对应的 NativeConfig

返回类型：NativeConfig

EnableGpuMultiStream()

开启线程流，目前的行为是为每一个线程绑定一个流，在将来该行为可能改变。

参数：

- None

返回：None

返回类型：void

thread_local_stream_enabled()

是否启用线程流。

参数：

- None

返回：是否启用线程流。

返回类型：bool

EnableMemoryOptim()

开启内/显存复用，具体降低内存效果取决于模型结构。

参数：

- None

返回：None

返回类型：void

enable_memory_optim()

是否开启内/显存复用。

参数：

- None

返回：是否开启内/显存复用。

返回类型：bool

EnableProfile()

打开 profile，运行结束后会打印所有 op 的耗时占比。

参数：

- None

返回：None

返回类型：void

profile_enabled()

是否开启 profile。

参数：

- None

返回：是否开启 profile

返回类型：bool

DisableGlogInfo()

去除 Paddle Inference 运行中的 log。

参数：

- None

返回：None

返回类型：void

glog_info_disabled()

是否禁用了 log。

参数：

- None

返回：是否禁用了 log

返回类型：bool

SetInValid()

设置 Config 为无效状态，仅内部使用，保证每一个 Config 仅用来初始化一次 Predictor。

参数：

- None

返回：None

返回类型：void

is_valid()

当前 Config 是否有效。

参数：

- None

返回：Config 是否有效

返回类型：bool

pass_builder()

返回 pass_builder，用来自定义图分析阶段选择的 ir。

示例：

```
Config config;
auto pass_builder = config.pass_builder()
pass_builder->DeletePass("fc_fuse_pass") // 去除 fc_fuse
```

参数：

- None

返回：pass_builder

返回类型：PassStrategy

PredictorPool

```
class PredictorPool;
```

PredictorPool 对 Predictor 进行了简单的封装，通过传入 config 和 thread 的数目来完成初始化，在每个线程中，根据自己的线程 id 直接从池中取出对应的 Predictor 来完成预测过程。

示例：

```
Config config;
// init config
int thread_num = 4;

PredictorPool pool(config, thread_num);

auto predictor0 = pool.Retrive(0);
...
auto predictor3 = pool.Retrive(3);
```

Retrive(idx)

根据线程 id 取出该线程对应的 Predictor。

参数：

- idx(int) - 线程 id

返回：线程对应的 Predictor

返回类型：Predictor*

C 预测 API 介绍

Fluid 提供了高度优化的C++ 预测库，为了方便使用，我们也提供了封装了 C++ 预测库对应的 C 接口。C 接口的使用方式，首先是需要 #include paddle_c_api.h，头文件 paddle_c_api.h 可以在 Paddle 的仓库中的 paddle/fluid/inference/capi/paddle_c_api.h 找到，或是在编译 Paddle 的 Paddle/build/路径下，build/fluid_inference_c_install_dir/paddle/include/路径下找到。此外，使用 CAPI 还需要在编译项目的时候，链接相关的编译的库 libpaddle_fluid_c.so。下面是详细的使用说明。

需要说明的是，与 C++ API 不同，C API 为了兼顾多语言封装的需要，将不会再设置默认参数，即使用时，所有的参数都需要用户显式地提供。

C 预测相关数据结构

使用 C 预测 API 与 C++ 预测 API 不完全一样，C 预测主要包括 PD_AnalysisConfig, PD_DataType, PD_Predictor, PD_Buffer 和 PD_ZeroCopyTensor。接下来将会进一步详细地介绍这些数据结构以及使用的方法，并提供相应的示例。

PD_AnalysisConfig

PD_AnalysisConfig 是创建预测引擎的配置，提供了模型路径设置、预测引擎运行设备选择以及多种优化预测流程的选项，主要包括以下方法

- PD_AnalysisConfig* PD_NewAnalysisConfig(): 新建一个 PD_AnalysisConfig 的指针。
- void PD_DeleteAnalysisConfig(PD_AnalysisConfig* config): 删除一个 PD_AnalysisConfig 的指针。
- void PD_SetModel(PD_AnalysisConfig* config, const char* model_dir, const char* params_path): 设置模型的路径，输入的参数包括 PD_AnalysisConfig，model_dir，params_path，其中 model_dir 是指的是模型保存位置的路径，一般不用包括文件名，params_path 为可选参数，注意:
  - 如果不给定 params_path，即 params_path 为 NULL，则认为该模型的参数存储路径与 model_dir 一致，且模型文件和参数文件是按照默认的文件名存储的，此时参数文件可能有多个。此时，需要用户输入参数与模型文件的 model_dir，即模型和参数保存的路径名，不需要指定文件名，同时，需要显式地设置 params_path 为 NULL。
  - 如果提供了 params_path，为了方便用户的自定义，则在指明 model_dir 路径最后需要加上模型文件的文件名传入，即 model_dir 传入对应的模型文件的路径，params_path 传入对应的模型参数文件的路径，需要指定文件名。
- const char* PD_ModelDir(const PD_AnalysisConfig* config): 如果未指明 PD_SetModel() 的 params_path，则可以返回模型文件夹路径。
- const char* PD_ProgFile(const PD_AnalysisConfig* config): 如果是指明 PD_SetModel() 的 params_path，则可以返回模型文件路径。
- const char* PD_ParamsFile(const PD_AnalysisConfig* config): 如果是指明 PD_SetModel() 的 params_path，则可以返回参数文件路径。
- void PD_SwitchSpecifyInputNames(PD_AnalysisConfig* config, bool x): 设置为 true 是指模型运算在读取输入的时候，依据名称来确定不同的输入，否则根据输入的顺序。使用 PD_ZeroCopyTensor 并且是多输入的情况，建议设置为 true。
- void PD_SwitchUseFeedFetchOps(PD_AnalysisConfig* config, bool x): 设置是否使用 feed，fetch op。在使用 PD_ZeroCopyTensor 必须设置该选项为 false。
- void PD_EnableUseGpu(PD_AnalysisConfig* config, uint64_t memory_pool_init_size_mb, int device_id): 设置开启 GPU，并且设定 GPU 显存 (单

位 M) 和设备的 Device ID。

- `void PD_DisableGpu(PD_AnalysisConfig* config)`: 禁用 GPU。
- `int PD_GpuDeviceId(const PD_AnalysisConfig* config)`: 返回使用的 GPU 设备的 ID。
- `void PD_SwitchIrOptim(PD_AnalysisConfig* config, bool x)`: 设置预测是否开启 IR 优化。
- `void PD_EnableTensorRtEngine(PD_AnalysisConfig* config, int workspace_size, int max_batch_size, int min_subgraph_size, Precision precision, bool use_static, bool use_calib_mode)`: 开启 TensorRT。关于参数的解释，详见使用 Paddle-TensorRT 库预测。
- `void PD_EnableMKLDNN(PD_AnalysisConfig* config)`: 开启 MKLDNN。

代码示例

首先，新建一个 `PD_AnalysisConfig` 的指针。

```
PD_AnalysisConfig* config = PD_NewAnalysisConfig();
```

如前文所述，设置模型和参数路径有两种形式：

- 当模型文件夹下存在一个以默认文件名保存的模型文件和多个参数文件时，传入模型文件夹路径，模型文件名默认为 `__model__`，需要显式地设置 `params_path` 为 `NULL`，不需要指定文件名。

```
const char* model_dir = "./model/";
PD_SetModel(config, model_dir, NULL);
```

- 当模型文件夹下只有一个模型文件和一个参数文件，传入模型文件和参数文件，需要指定文件名。

```
const char* model_path = "./model/model";
const char* params_path = "./params/params";
PD_SetModel(config, model_path, params_path);
```

其他预测引擎配置选项示例如下

```
PD_EnableUseGpu(config, 100, 0); // 初始化 100M 显存，使用的 gpu id 为 0
PD_GpuDeviceId(config);          // 返回正在使用的 gpu id
PD_DisableGpu(config);           // 禁用 gpu
PD_SwitchIrOptim(config, true);  // 开启 IR 优化
PD_EnableMKLDNN(config);         // 开启 MKLDNN
PD_SwitchSpecifyInputNames(config, true);
PD_SwitchUseFeedFetchOps(config, false);
```

PD_ZeroCopyTensor

PD_ZeroCopyTensor 是设置数据传入预测运算的数据结构。包括一下成员：

- data - (PD_Buffer)：设置传入数据的值。
- shape - (PD_Buffer)：设置传入数据的形状（shape）。
- lod - (PD_Buffer)：设置数据的 lod，目前只支持一阶的 lod。
- dtype - (PD_DataType)：设置传入数据的数据类型，用枚举 PD_DataType 表示。
- name - (char*)：设置传入数据的名称。

涉及使用 PD_ZeroCopyTensor 有以下方法：

- PD_ZeroCopyTensor* PD_NewZeroCopyTensor()：新创建一个 PD_ZeroCopyTensor 的指针。
- void PD_DeleteZeroCopyTensor(PD_ZeroCopyTensor*)：删除一个 PD_ZeroCopyTensor 的指针。
- void PD_InitZeroCopyTensor(PD_ZeroCopyTensor*)：使用默认初始化一个 PD_ZeroCopyTensor 的指针并分配的内存空间。
- void PD_DestroyZeroCopyTensor(PD_ZeroCopyTensor*)：删除 PD_ZeroCopyTensor 指针中，data，shape，lod 的 PD_Buffer 的变量。

PD_DataType

PD_DataType 是一个提供给用户的枚举，用于设定存有用户数据的 PD_ZeroCopyTensor 的数据类型。包括以下成员：

- PD_FLOAT32: 32 位浮点型
- PD_INT32: 32 位整型
- PD_INT64: 64 位整型
- PD_UINT8: 8 位无符号整型

代码示例

首先可以新建一个 PD_ZeroCopyTensor。

```
PD_ZeroCopyTensor input;
PD_InitZeroCopyTensor(&input);
```

调用设置 PD_ZeroCopyTensor 的数据类型的方式如下：

```
input.dtype = PD_FLOAT32;
```

PD_Buffer

PD_Buffer 可以用于设置 PD_ZeroCopyTensor 数据结构中，数据的 data，shape 和 lod。包括以下成员：

- data: 输入的数据，类型是 void*，用于存储数据开始的地址。
- length: 输入数据的实际的字节长度。
- capacity: 为数据分配的内存大小，必定大于等于 length。

示例代码

```
PD_ZeroCopyTensor input;
PD_InitZeroCopyTensor(&input);
// 设置输入的名称
input.name = "data";
// 设置输入的数据大小
input.data.capacity = sizeof(float) * 1 * 3 * 300 * 300;
input.data.length = input.data.capacity;
input.data.data = malloc(input.data.capacity);
// 设置数据的输入的形状 shape
int shape[] = {1, 3, 300, 300};
input.shape.data = (int *)shape;
input.shape.capacity = sizeof(shape);
input.shape.length = sizeof(shape);
// 设置输入数据的类型
input.dtype = PD_FLOAT32;
```

PD_Predictor

PD_Predictor 是一个高性能预测引擎，该引擎通过对计算图的分析，可以完成对计算图的一系列的优化（如 OP 的融合、内存/显存的优化、MKLDNN，TensorRT 等底层加速库的支持等）。主要包括一下函数：

- PD_Predictor* PD_NewPredictor(const PD_AnalysisConfig* config)：创建一个新的 PD_Predictor 的指针。
- void PD_DeletePredictor(PD_Predictor* predictor)：删除一个 PD_Predictor 的指针。
- int PD_GetInputNum(const PD_Predictor* predictor)：获取模型输入的个数。

- `int PD_GetOutputNum(const PD_Predictor* predictor)`: 获取模型输出的个数。
- `const char* PD_GetInputName(const PD_Predictor* predictor, int n)`: 获取模型第 `n` 个输入的名称。
- `const char* PD_GetOutputName(const PD_Predictor* predictor, int n)`: 获取模型第 `n` 个输出的名称。
- `void PD_SetZeroCopyInput(PD_Predictor* predictor, const PD_ZeroCopyTensor* tensor)`: 使用 `PD_ZeroCopyTensor` 数据结构设置模型输入的具体值、形状、lod 等信息。目前只支持一阶 lod。
- `void PD_GetZeroCopyOutput(PD_Predictor* predictor, PD_ZeroCopyTensor* tensor)`: 使用 `PD_ZeroCopyTensor` 数据结构获取模型输出的具体值、形状、lod 等信息。目前只支持一阶 lod。
- `void PD_ZeroCopyRun(PD_Predictor* predictor)`: 运行预测的引擎，完成模型由输入到输出的计算。

代码示例

如前文所述，当完成网络配置 `PD_AnalysisConfig` 以及输入 `PD_ZeroCopyTensor` 的设置之后，只需要简单的几行代码就可以获得模型的输出。

首先完成 `PD_AnalysisConfig` 的设置，设置的方式与相关的函数如前文所述，这里同样给出了示例。

```
PD_AnalysisConfig* config = PD_NewAnalysisConfig();
const char* model_dir = "./model/";
PD_SetModel(config, model_dir, NULL);
PD_DisableGpu(config);
PD_SwitchSpecifyInputNames(config, true); // 使用 PD_ZeroCopyTensor 并且是多输入建议设置。
PD_SwitchUseFeedFetchOps(config, false);  // 使用 PD_ZeroCopyTensor 一定需要设置为 false。
```

其次，完成相应的输入的设置，设置的方式如前文所述，这里同样给出了示例。

```
PD_ZeroCopyTensor input;
PD_InitZeroCopyTensor(&input);
// 设置输入的名称
input.name = (char *)(PD_GetInputName(predictor, 0));
// 设置输入的数据大小
input.data.capacity = sizeof(float) * 1 * 3 * 300 * 300;
input.data.length = input.data.capacity;
input.data.data = malloc(input.data.capacity);
// 设置数据的输入的形状 (shape)
int shape[] = {1, 3, 300, 300};
input.shape.data = (int *)shape;
```

(下页继续)

(续上页)

```
input.shape.capacity = sizeof(shape);
input.shape.length = sizeof(shape);
// 设置输入数据的类型
input.dtype = PD_FLOAT32;
```

最后，执行预测引擎，完成计算的步骤。

```
PD_Predictor *predictor = PD_NewPredictor(config);

int input_num = PD_GetInputNum(predictor);
printf("Input num: %d\n", input_num);
int output_num = PD_GetOutputNum(predictor);
printf("Output num: %d\n", output_num);

PD_SetZeroCopyInput(predictor, &input); // 这里只有一个输入，根据多输入情况，可以传入一个数组

PD_ZeroCopyRun(predictor); // 执行预测引擎

PD_ZeroCopyTensor output;
PD_InitZeroCopyTensor(&output);
output.name = (char *)(PD_GetOutputName(predictor, 0));
PD_GetZeroCopyOutput(predictor, &output);
```

最后，可以根据前文所述的 `PD_ZeroCopyTensor` 的数据结构，获得返回的数据的值等信息。

完整使用示例

下面是使用 Fluid C API 进行预测的一个完整示例，使用 resnet50 模型

下载resnet50 模型并解压，运行如下代码将会调用预测引擎。

```
#include "paddle_c_api.h"
#include <memory.h>
#include <malloc.h>

/*
 * The main procedures to run a predictor according to c-api:
 * 1. Create config to set how to process the inference.
 * 2. Prepare the input PD_ZeroCopyTensor for the inference.
 * 3. Set PD_Predictor.
 * 4. Call PD_ZeroCopyRun() to start.
 * 5. Obtain the output.
 * 6. According to the size of the PD_PaddleBuf's data's size, print all the output␣
↪data.
```

(下页继续)

(续上页)

```
 */
int main() {
    // 配置 PD_AnalysisConfig
    PD_AnalysisConfig* config = PD_NewAnalysisConfig();
    PD_DisableGpu(config);
    const char* model_path = "./model/model";
    const char* params_path = "./model/params";
    PD_SetModel(config, model_path, params_path);
    PD_SwitchSpecifyInputNames(config, true);
    PD_SwitchUseFeedFetchOps(config, false);

    // 新建一个 PD_Predictor 的指针
    PD_Predictor *predictor = PD_NewPredictor(config);
    // 获取输入输出的个数
    int input_num = PD_GetInputNum(predictor);
    printf("Input num: %d\n", input_num);
    int output_num = PD_GetOutputNum(predictor);
    printf("Output num: %d\n", output_num);

    // 设置输入的数据结构
    PD_ZeroCopyTensor input;
    PD_InitZeroCopyTensor(&input);
    // 设置输入的名称
    input.name = (char *)(PD_GetInputName(predictor, 0));
    // 设置输入的数据大小
    input.data.capacity = sizeof(float) * 1 * 3 * 318 * 318;
    input.data.length = input.data.capacity;
    input.data.data = malloc(input.data.capacity);
    memset(input.data.data, 0, (sizeof(float) * 3 * 318 * 318));

    // 设置数据的输入的形状 (shape)
    int shape[] = {1, 3, 318, 318};
    input.shape.data = (int *)shape;
    input.shape.capacity = sizeof(shape);
    input.shape.length = sizeof(shape);
    // 设置输入数据的类型
    input.dtype = PD_FLOAT32;

    PD_SetZeroCopyInput(predictor, &input);

    // 执行预测引擎
    PD_ZeroCopyRun(predictor);
```

(下页继续)

(续上页)

```
    // 获取预测输出
    PD_ZeroCopyTensor output;
    PD_InitZeroCopyTensor(&output);
    output.name = (char *)(PD_GetOutputName(predictor, 0));
    // 获取 output 之后，可以通过该数据结构，读取到 data, shape 等信息
    PD_GetZeroCopyOutput(predictor, &output);

    float* result = (float *)(output.data.data);
    int result_length = output.data.length / sizeof(float);

    return 0;
}
```

运行以上代码，需要将 paddle_c_api.h 拷贝到指定位置，确保编译时可以找到这个头文件。同时，需要将 libpaddle_fluid_c.so 的路径加入环境变量。

最后可以使用 gcc 命令编译。

```
gcc ${SOURCE_NAME} \
    -lpaddle_fluid_c
```

Python 预测 API 介绍

Paddle 提供了高度优化的C++ 预测库，为了方便使用，我们也提供了 C++ 预测库对应的 Python 接口，下面是详细的使用说明。

如果您在使用 2.0 之前的 Paddle，请参考旧版 API文档。

Python 预测相关数据结构

使用 Python 预测 API 与 C++ 预测 API 相似，主要包括 `Tensor`, `DataType`, `Config` 和 `Predictor`，分别对应于 C++ API 中同名的类型。

DataType

class paddle.inference.DataType

`DataType` 定义了 `Tensor` 的数据类型，由传入 `Tensor` 的 numpy 数组类型确定，包括以下成员

- `INT64`: 64 位整型
- `INT32`: 32 位整型
- `FLOAT32`: 32 位浮点型

PrecisionType

class paddle.3.inference.PrecisionType

`PrecisionType` 定义了 `Predictor` 运行的精度模式，包括一下成员

- `Float32`: fp32 模式运行
- `Half`: fp16 模式运行
- `Int8`: int8 模式运行

Tensor

class paddle.inference.Tensor

`Tensor` 是 `Predictor` 的一种输入/输出数据结构，通过 `predictor` 获取输入/输出 handle 得到，主要提供以下方法

- `copy_from_cpu`: 从 cpu 获取模型运行所需输入数据
- `copy_to_cpu`: 获取模型运行输出结果
- `lod`: 获取 lod 信息
- `set_lod`: 设置 lod 信息
- `shape`: 获取 shape 信息
- `reshape`: 设置 shape 信息
- `type`: 获取 DataType 信息

```
# 创建 predictor
predictor = create_predictor(config)

# 获取输入的名称
input_names = predictor.get_input_names()
input_tensor = predictor.get_input_handle(input_names[0])

# 设置输入
fake_input = numpy.random.randn(1, 3, 318, 318).astype("float32")
input_tensor.copy_from_cpu(fake_input)

# 运行 predictor
predictor.run()

# 获取输出
output_names = predictor.get_output_names()
```

(下页继续)

(续上页)

```
output_tensor = predictor.get_output_handle(output_names[0])
output_data = output_tensor.copy_to_cpu() # numpy.ndarray 类型
```

Config

class paddle.inference.Config

Config 是创建预测引擎的配置，提供了模型路径设置、预测引擎运行设备选择以及多种优化预测流程的选项，主要包括以下方法

- set_model: 设置模型的路径
- model_dir: 返回模型文件夹路径
- prog_file: 返回模型文件路径
- params_file: 返回参数文件路径
- enable_use_gpu: 设置 GPU 显存 (单位 M) 和 Device ID
- disable_gpu: 禁用 GPU
- gpu_device_id: 返回使用的 GPU ID
- switch_ir_optim: IR 优化 (默认开启)
- enable_tensorrt_engine: 开启 TensorRT
- enable_mkldnn: 开启 MKLDNN
- disable_glog_info: 禁用预测中的 glog 日志
- delete_pass: 预测的时候删除指定的 pass

代码示例

设置模型和参数路径有两种形式：

- 当模型文件夹下存在一个模型文件和多个参数文件时，传入模型文件夹路径，模型文件名默认为 __model__

```
config = Config("./model")
```

- 当模型文件夹下只有一个模型文件和一个参数文件时，传入模型文件和参数文件路径

```
config = Config("./model/model", "./model/params")
```

使用 set_model 方法设置模型和参数路径方式同上

其他预测引擎配置选项示例如下

```
config.enable_use_gpu(100, 0) # 初始化 100M 显存，使用 gpu id 为 0
config.gpu_device_id()        # 返回正在使用的 gpu id
config.disable_gpu()          # 禁用 gpu
config.switch_ir_optim(True)  # 开启 IR 优化
config.enable_tensorrt_engine(precision_mode=PrecisionType.Float32,
                              use_calib_mode=True) # 开启 TensorRT 预测，精度为 fp32，开启
int8 离线量化
config.enable_mkldnn()          # 开启 MKLDNN
```

Predictor

class paddle.inference.Predictor

Predictor 是运行预测的引擎，由 paddle.inference.create_predictor(config) 创建，主要提供以下方法

- run(): 运行预测引擎，返回预测结果
- get_input_names(): 获取输入的名称
- get_input_handle(input_name: str): 根据输入的名称获取对应的 Tensor
- get_output_names(): 获取输出的名称
- get_output_handle(output_name: str): 根据输出的名称获取对应的 Tensor

代码示例

```
# 设置完 AnalysisConfig 后创建预测引擎 PaddlePredictor
predictor = create_predictor(config)

# 获取输入的名称
input_names = predictor.get_input_names()
input_handle = predictor.get_input_handle(input_names[0])

# 设置输入
fake_input = numpy.random.randn(1, 3, 318, 318).astype("float32")
input_handle.reshape([1, 3, 318, 318])
input_handle.copy_from_cpu(fake_input)

# 运行 predictor
predictor.run()

# 获取输出
```

(下页继续)

(续上页)

```
output_names = predictor.get_output_names()
output_handle = predictor.get_output_handle(output_names[0])
```

完整使用示例

下面是使用 Paddle Inference Python API 进行预测的一个完整示例，使用 resnet50 模型

下载resnet50 模型并解压，运行如下命令将会调用预测引擎

```
python resnet50_infer.py --model_file ./model/model --params_file ./model/params --
↪batch_size 2
```

resnet50_infer.py 的内容是

```
import argparse
import numpy as np
from paddle.inference import Config
from paddle.inference import create_predictor


def main():
    args = parse_args()

    # 设置 AnalysisConfig
    config = set_config(args)

    # 创建 PaddlePredictor
    predictor = create_predictor(config)

    # 获取输入的名称
    input_names = predictor.get_input_names()
    input_handle = predictor.get_input_handle(input_names[0])

    # 设置输入
    fake_input = np.random.randn(1, 3, 318, 318).astype("float32")
    input_handle.reshape([1, 3, 318, 318])
    input_handle.copy_from_cpu(fake_input)

    # 运行 predictor
    predictor.run()

    # 获取输出
    output_names = predictor.get_output_names()
```

(下页继续)

(续上页)

```
    output_handle = predictor.get_output_handle(output_names[0])
    output_data = output_handle.copy_to_cpu() # numpy.ndarray 类型


def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--model_file", type=str, help="model filename")
    parser.add_argument("--params_file", type=str, help="parameter filename")
    parser.add_argument("--batch_size", type=int, default=1, help="batch size")

    return parser.parse_args()


def set_config(args):
    config = Config(args.model_file, args.params_file)
    config.disable_gpu()
    config.switch_use_feed_fetch_ops(False)
    config.switch_specify_input_names(True)
    return config


if __name__ == "__main__":
    main()
```

支持方法列表

- Tensor
  - copy_from_cpu(input: numpy.ndarray) -> None
  - copy_to_cpu() -> numpy.ndarray
  - reshape(input: numpy.ndarray|List[int]) -> None
  - shape() -> List[int]
  - set_lod(input: numpy.ndarray|List[List[int]]) -> None
  - lod() -> List[List[int]]
  - type() -> PaddleDType
- Config
  - set_model(model_dir: str) -> None
  - set_model(prog_file: str, params_file: str) -> None

- set_model_buffer(model: str, model_size: int, param: str, param_size: int) -> None
- model_dir() -> str
- prog_file() -> str
- params_file() -> str
- model_from_memory() -> bool
- set_cpu_math_library_num_threads(num: int) -> None
- enable_use_gpu(memory_pool_init_size_mb: int, device_id: int) -> None
- use_gpu() -> bool
- gpu_device_id() -> int
- switch_ir_optim(x: bool = True) -> None
- switch_ir_debug(x: int=True) -> None
- ir_optim() -> bool
- enable_tensorrt_engine(workspace_size: int = 1 << 20, max_batch_size: int, min_subgraph_size: int, precision_mode: AnalysisConfig.precision, use_static: bool, use_calib_mode: bool) -> None
- set_trt_dynamic_shape_info(min_input_shape: Dict[str, List[int]]={}, max_input_shape: Dict[str, List[int]]={}, optim_input_shape: Dict[str, List[int]]={}, disable_trt_plugin_fp16: bool=False) -> None
- tensorrt_engine_enabled() -> bool
- enable_mkldnn() -> None
- enable_mkldnn_bfloat16() -> None
- mkldnn_enabled() -> bool
- set_mkldnn_cache_capacity(capacity: int=0) -> None
- set_mkldnn_op(ops: Set[str]) -> None
- set_optim_cache_dir(dir: str) -> None
- disable_glog_info() -> None
- pass_builder() -> paddle::PassStrategy
- delete_pass(pass_name: str) -> None
- cpu_math_library_num_threads() -> int
- disable_gpu() -> None

  - enable_lite_engine(precision: PrecisionType, zero_copy: bool, passes_filter: List[str]=[], ops_filter: List[str]=[]) -> None
  - lite_engine_enabled() -> bool
  - enable_memory_optim() -> None
  - enable_profile() -> None
  - enable_quantizer() -> None
  - quantizer_config() -> paddle::MkldnnQuantizerConfig
  - fraction_of_gpu_memory_for_pool() -> float
  - memory_pool_init_size_mb() -> int
  - glog_info_disabled() -> bool
  - gpu_device_id() -> int
  - specify_input_name() -> bool
  - switch_specify_input_names(x: bool=True) -> None
  - specify_input_name(q) -> bool
  - switch_use_feed_fetch_ops(x: int=True) -> None
  - use_feed_fetch_ops_enabled() -> bool
  - to_native_config() -> paddle.fluid.core_avx.NativeConfig
- create_predictor(config: Config) -> Predictor
- Predictor
  - run() -> None
  - get_input_names() -> List[str]
  - get_input_handle(input_name: str) -> Tensor
  - get_output_names() -> List[str]
  - get_output_handle(output_name: str) -> Tensor
  - clear_intermediate_tensor() -> None
  - try_shrink_memory() -> None
  - clone() -> Predictor
- PredictorPool
  - retrive(idx: int) -> Predictor

可参考对应的C++ 预测接口，其中定义了每个接口的参数和返回值

5.1.2 移动端部署

本模块介绍了飞桨的端侧推理引擎 Paddle-Lite：

- Paddle Lite：简要介绍了 Paddle-Lite 特点以及使用说明。

Paddle-Lite

Paddle-Lite 为 Paddle-Mobile 的升级版，定位支持包括手机移动端在内更多场景的轻量化高效预测，支持更广泛的硬件和平台，是一个高性能、轻量级的深度学习预测引擎。在保持和 PaddlePaddle 无缝对接外，也兼容支持其他训练框架产出的模型。

完整使用文档位于 Paddle-Lite 文档 。

特性

轻量级

执行阶段和计算优化阶段实现良好解耦拆分，移动端可以直接部署执行阶段，无任何第三方依赖。包含完整的 80 个 Op+85 个 Kernel 的动态库，对于 ARMV7 只有 800K，ARMV8 下为 1.3M，并可以裁剪到更低。在应用部署时，载入模型即可直接预测，无需额外分析优化。

高性能

极致的 ARM CPU 性能优化，针对不同微架构特点实现 kernel 的定制，最大发挥计算性能，在主流模型上展现出领先的速度优势。支持量化模型，结合PaddleSlim 模型压缩工具 中量化功能，可以提供高精度高性能的预测能力。在 Huawei NPU，FPGA 上也具有有很好的性能表现。

最新性能数据位于 Benchmark 文档。

通用性

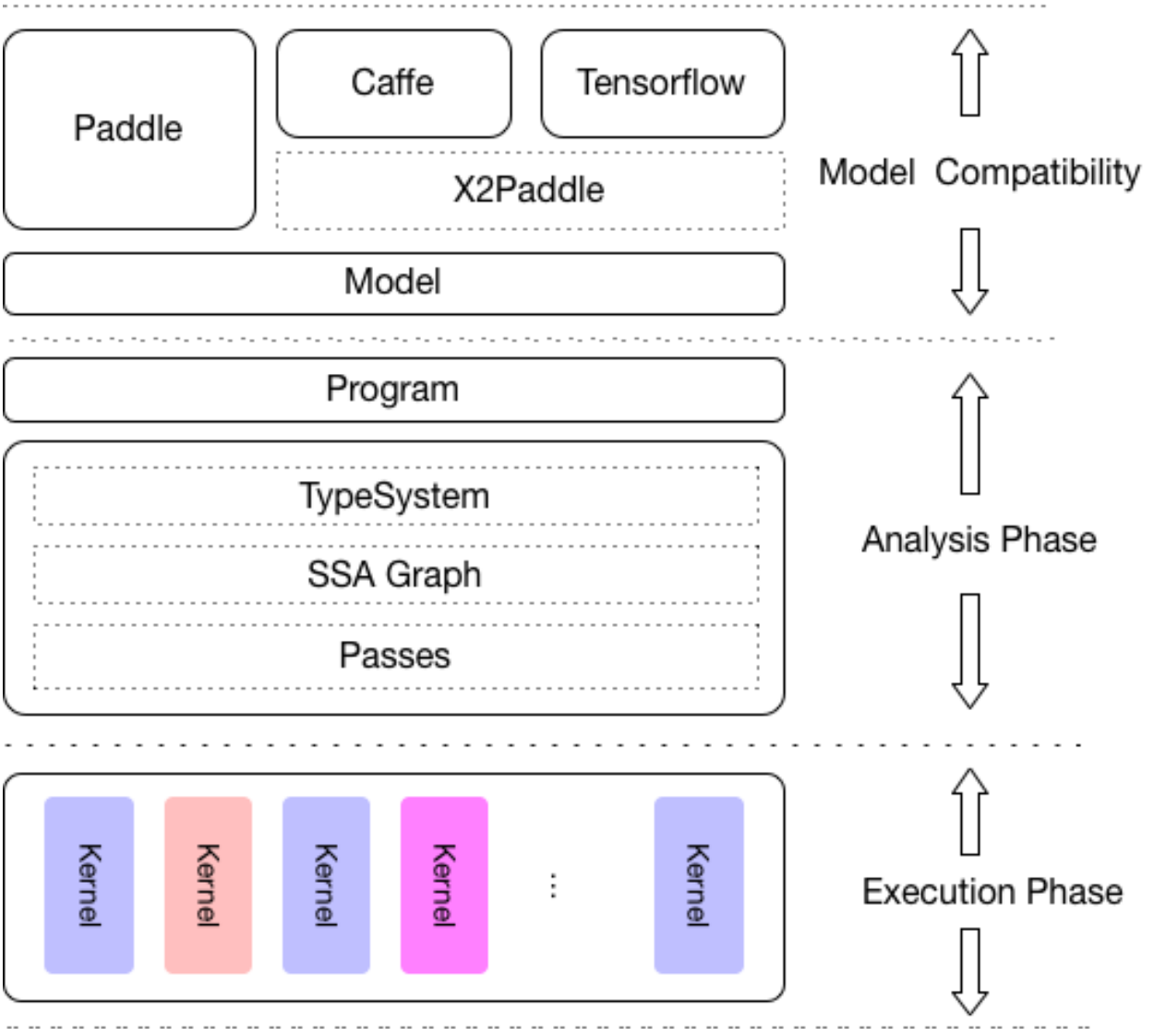
硬件方面，Paddle-Lite 的架构设计为多硬件兼容支持做了良好设计。除了支持 ARM CPU、Mali GPU、Adreno GPU，还特别支持了华为 NPU，以及 FPGA 等边缘设备广泛使用的硬件。即将支持支持包括寒武纪、比特大陆等 AI 芯片，未来会增加对更多硬件的支持。

模型支持方面，Paddle-Lite 和 PaddlePaddle 训练框架的 Op 对齐，提供更广泛的模型支持能力。目前已严格验证 18 个模型 85 个 OP 的精度和性能，对视觉类模型做到了较为充分的支持，覆盖分类、检测和定位，包含了特色的 OCR 模型的支持。未来会持续增加更多模型的支持验证。

框架兼容方面：除了 PaddlePaddle 外，对其他训练框架也提供兼容支持。当前，支持 Caffe 和 TensorFlow 训练出来的模型，通过 [X2Paddle] (https://github.com/PaddlePaddle/X2Paddle) 转换工具实现。接下来将会对 ONNX 等格式模型提供兼容支持。

架构

Paddle-Lite 的架构设计着重考虑了对多硬件和平台的支持，并且强化了多个硬件在一个模型中混合执行的能力，多个层面的性能优化处理，以及对端侧应用的轻量化设计。



其中，Analysis Phase 包括了 MIR(Machine IR) 相关模块，能够对原有的模型的计算图针对具体的硬件列表进行算子融合、计算裁剪在内的多种优化。Execution Phase 只涉及到 Kernel 的执行，且可以单独部署，以支持极致的轻量级部署。

Paddle-Mobile 升级为 Paddle-Lite 的说明

原 Paddle-Mobile 作为一个致力于嵌入式平台的 PaddlePaddle 预测引擎，已支持多种硬件平台，包括 ARM CPU、Mali GPU、Adreno GPU，以及支持苹果设备的 GPU Metal 实现、ZU5、ZU9 等 FPGA 开发板、树莓派等 arm-linux 开发板。在百度内已经过广泛业务场景应用验证。对应设计文档可参考: mobile/README

Paddle-Mobile 整体升级重构并更名为 Paddle-Lite 后，原 paddle-mobile 的底层能力大部分已集成到新架构 下。作为过渡，暂时保留原 Paddle-mobile 代码。主体代码位于 `mobile/` 目录中，后续一段时间会继续维护，并完成全部迁移。新功能会统一到新架构 下开发。

metal, web 的模块相对独立，会继续在 `./metal` 和 `./web` 目录下开发和维护。对苹果设备的 GPU Metal 实现的需求及 web 前端预测需求，可以直接进入这两个目录。

致谢

Paddle-Lite 借鉴了以下开源项目：

- ARM compute library
- Anakin ，Anakin 对应底层的一些优化实现已被集成到 Paddle-Lite。Anakin 作为 PaddlePaddle 组织下的一个高性能预测项目，极具前瞻性，对 Paddle-Lite 有重要贡献。Anakin 已和本项目实现整合。之后，Anakin 不再升级。

交流与反馈

- 欢迎您通过 Github Issues 来提交问题、报告与建议
- 微信公众号：飞桨 PaddlePaddle
- QQ 群: 696965088
- 论坛: 欢迎大家在PaddlePaddle 论坛分享在使用 PaddlePaddle 中遇到的问题和经验, 营造良好的论坛氛围

5.1.3 模型压缩

PaddleSlim 是一个模型压缩工具库，包含模型剪裁、定点量化、知识蒸馏、超参搜索和模型结构搜索等一系列模型压缩策略。

对于业务用户，PaddleSlim 提供完整的模型压缩解决方案，可用于图像分类、检测、分割等各种类型的视觉场景。同时也在持续探索 NLP 领域模型的压缩方案。另外，PaddleSlim 提供且在不断完善各种压缩策略在经典开源任务的 benchmark, 以便业务用户参考。

对于模型压缩算法研究者或开发者，PaddleSlim 提供各种压缩策略的底层辅助接口，方便用户复现、调研和使用最新论文方法。PaddleSlim 会从底层能力、技术咨询合作和业务场景等角度支持开发者进行模型压缩策略相关的创新工作。

功能

- 模型剪裁
  - 卷积通道均匀剪裁
  - 基于敏感度的卷积通道剪裁
  - 基于进化算法的自动剪裁
- 定点量化
  - 在线量化训练（training aware）
  - 离线量化（post training）
- 知识蒸馏
  - 支持单进程知识蒸馏
  - 支持多进程分布式知识蒸馏
- 神经网络结构自动搜索（NAS）
  - 支持基于进化算法的轻量神经网络结构自动搜索
  - 支持 One-Shot 网络结构自动搜索
  - 支持 FLOPS / 硬件延时约束
  - 支持多平台模型延时评估
  - 支持用户自定义搜索算法和搜索空间

安装

依赖：

Paddle >= 1.7.0

```
pip install paddleslim -i https://pypi.org/simple
```

使用

- 快速开始：通过简单示例介绍如何快速使用 PaddleSlim。
- 进阶教程：PaddleSlim 高阶教程。
- 模型库：各个压缩策略在图像分类、目标检测和图像语义分割模型上的实验结论，包括模型精度、预测速度和可供下载的预训练模型。
- API 文档
- Paddle 检测库：介绍如何在检测库中使用 PaddleSlim。

- Paddle 分割库：介绍如何在分割库中使用 PaddleSlim。
- PaddleLite：介绍如何使用预测库 PaddleLite 部署 PaddleSlim 产出的模型。

部分压缩策略效果

分类模型

数据: ImageNet2012; 模型: MobileNetV1;

图像检测模型

数据：Pascal VOC；模型：MobileNet-V1-YOLOv3

数据：COCO；模型：MobileNet-V1-YOLOv3

搜索

数据：ImageNet2012; 模型：MobileNetV2

CHAPTER 6

分布式训练

您可以通过以下内容, 了解飞桨分布式训练的特性和使用指南:

- 分布式训练快速开始 : 使用飞桨框架快速开始分布式训练。
- 使用 FleetAPI 进行分布式训练 : 使用飞桨框架 FleetAPI 完成分布式训练。

6.1 分布式训练快速开始

6.1.1 使用 Fleet API 进行分布式训练

从 PaddlePaddle Release 1.5.1 开始，官方推荐使用 Fleet API 进行分布式训练。

准备条件

- [x] 成功安装 PaddlePaddle，如果尚未安装，请参考 快速开始
- [x] 学会最基本的单机训练方法，请参考 单机训练 中描述的单机训练，进行学习

点击率预估任务

本文使用一个简单的示例，点击率预估任务，来说明如何使用 Fleet API 进行分布式训练的配置方法，并利用单机环境模拟分布式环境给出运行示例。

为了方便学习，这里给出的示例是单机与多机混合的代码，用户可以通过不同的启动命令进行单机或多机任务的启动。

```
from __future__ import print_function
from args import parse_args
import os
import sys

import paddle
import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

from network_conf import ctr_dnn_model_dataset

dense_feature_dim = 13
sparse_feature_dim = 10000001

batch_size = 100
thread_num = 10
embedding_size = 10

args = parse_args()

def main_function(is_local):

    # common code for local training and distributed training
    dense_input = paddle.static.data(
      name="dense_input", shape=[dense_feature_dim], dtype='float32')
    sparse_input_ids = [
          paddle.static.data(name="C" + str(i), shape=[1], lod_level=1,
                             dtype="int64") for i in range(1, 27)]

    label = paddle.static.data(name="label", shape=[1], dtype="int64")

    dataset = paddle.distributed.QueueDataset()
    dataset.init(
          batch_size=batch_size,
          thread_num=thread_num,
          input_type=0,
          pipe_command=python criteo_reader.py %d" % sparse_feature_dim,
```

(下页继续)

(续上页)

```
        use_var=[dense_input] + sparse_input_ids + [label])

    whole_filelist = ["raw_data/part-%d" % x
                      for x in range(len(os.listdir("raw_data")))]
    dataset.set_filelist(whole_filelist)

    loss, auc_var, batch_auc_var = ctr_dnn_model_dataset(
        dense_input, sparse_input_ids, label, embedding_size,
        sparse_feature_dim)

    exe = paddle.static.Executor(paddle.CPUPlace())

    def train_loop(epoch=20):
        for i in range(epoch):
            exe.train_from_dataset(program=paddle.static.default_main_program(),
                                   dataset=dataset,
                                   fetch_list=[auc_var],
                                   fetch_info=["auc"],
                                   debug=False)

    # local training
    def local_train():
        optimizer = paddle.optimizer.SGD(learning_rate=1e-4)
        optimizer.minimize(loss)
        exe.run(paddle.static.default_startup_program())
        train_loop()

# distributed training
    def dist_train():
        role = role_maker.PaddleCloudRoleMaker()
        fleet.init(role)

        strategy = paddle.distributed.fleet.DistributedStrategy()
        strategy.a_sync = True
        optimizer = paddle.optimizer.SGD(learning_rate=1e-4)
        optimizer = fleet.distributed_optimizer(optimizer, strategy)
        optimizer.minimize(loss)

        if fleet.is_server():
            fleet.init_server()
            fleet.run_server()

        elif fleet.is_worker():
```

(下页继续)

(续上页)

```
            fleet.init_worker()
            exe.run(paddle.static.default_startup_program())
            train_loop()

    if is_local:
        local_train()
    else:
        dist_train()

if __name__ == '__main__':
    main_function(args.is_local)
```

- 说明：示例中使用的 IO 方法是 dataset，想了解具体的文档和用法请参考 Dataset API 。

单机训练启动命令

```
python train.py --is_local 1
```

单机模拟分布式训练的启动命令

在单机模拟多机训练的启动命令，这里我们用到了 paddle 内置的一个启动器 launch_ps，用户可以指定 worker 和 server 的数量进行参数服务器任务的启动

```
python -m paddle.distributed.launch_ps --worker_num 2 --server_num 2 train.py
```

任务运行的日志在工作目录的 logs 目录下可以查看，当您能够使用单机模拟分布式训练。

6.2 使用 FleetAPI 进行分布式训练

6.2.1 FleetAPI 设计说明

Fleet 是 PaddlePaddle 分布式训练的高级 API。Fleet 的命名出自于 PaddlePaddle，象征一个舰队中的多只双桨船协同工作。Fleet 的设计在易用性和算法可扩展性方面做出了权衡。用户可以很容易从单机版的训练程序，通过添加几行代码切换到分布式训练程序。此外，分布式训练的算法也可以通过 Fleet API 接口灵活定义。

6.2.2 Fleet API 快速上手示例

下面会针对 Fleet API 最常见的两种使用场景，用一个模型做示例，目的是让用户有快速上手体验的模板。

- 假设我们定义 MLP 网络如下：

```
import paddle

def mlp(input_x, input_y, hid_dim=128, label_dim=2):
  fc_1 = paddle.static.nn.fc(input=input_x, size=hid_dim, act='tanh')
  fc_2 = paddle.static.nn.fc(input=fc_1, size=hid_dim, act='tanh')
  prediction = paddle.static.nn.fc(input=[fc_2], size=label_dim, act='softmax')
  cost = paddle.static.nn.cross_entropy(input=prediction, label=input_y)
  avg_cost = paddle.static.nn.mean(x=cost)
  return avg_cost
```

- 定义一个在内存生成数据的 Reader 如下：

```
import numpy as np

def gen_data():
    return {"x": np.random.random(size=(128, 32)).astype('float32'),
            "y": np.random.randint(2, size=(128, 1)).astype('int64')}
```

- 单机 Trainer 定义

```
import paddle
from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)
optimizer.minimize(cost)
place = paddle.CUDAPlace(0)

exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())
step = 1001
for i in range(step):
  cost_val = exe.run(feed=gen_data(), fetch_list=[cost.name])
  print("step%d cost=%f" % (i, cost_val[0]))
```

- Parameter Server 训练方法

参数服务器方法对于大规模数据，简单模型的并行训练非常适用，我们基于单机模型的定义给出使用 Parameter Server 进行训练的示例如下：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)

role = role_maker.PaddleCloudRoleMaker()
fleet.init(role)

strategy = paddle.distributed.fleet.DistributedStrategy()
strategy.a_sync = True

optimizer = fleet.distributed_optimizer(optimizer, strategy)
optimizer.minimize(cost)

if fleet.is_server():
  fleet.init_server()
  fleet.run_server()

elif fleet.is_worker():
  place = paddle.CPUPlace()
  exe = paddle.static.Executor(place)
  exe.run(paddle.static.default_startup_program())

  step = 1001
  for i in range(step):
    cost_val = exe.run(
        program=paddle.static.default_main_program(),
        feed=gen_data(),
        fetch_list=[cost.name])
    print("worker_index: %d, step%d cost = %f" %
         (fleet.worker_index(), i, cost_val[0]))
```

- Collective 训练方法

  Collective Training 通常在 GPU 多机多卡训练中使用，一般在复杂模型的训练中比较常见，我们基于上面的单机模型定义给出使用 Collective 方法进行分布式训练的示例如下：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)
role = role_maker.PaddleCloudRoleMaker(is_collective=True)
fleet.init(role)

optimizer = fleet.distributed_optimizer(optimizer)
optimizer.minimize(cost)
place = paddle.CUDAPlace(0)

exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())

step = 1001
for i in range(step):
  cost_val = exe.run(
      program=paddle.static.default_main_program(),
      feed=gen_data(),
      fetch_list=[cost.name])
  print("worker_index: %d, step%d cost = %f" %
        (fleet.worker_index(), i, cost_val[0]))
```

6.2.3 Fleet API 相关的接口说明

Fleet API 接口

- init(role_maker=None)
  - fleet 初始化，需要在使用 fleet 其他接口前先调用，用于定义多机的环境配置
- is_worker()
  - Parameter Server 训练中使用，判断当前节点是否是 Worker 节点，是则返回 True，否则返回 False
- is_server(model_dir=None)
  - Parameter Server 训练中使用，判断当前节点是否是 Server 节点，是则返回 True，否则返回 False
- init_server()
  - Parameter Server 训练中，fleet 加载 model_dir 中保存的模型相关参数进行 parameter server 的初始化
- run_server()
  - Parameter Server 训练中使用，用来启动 server 端服务
- init_worker()
  - Parameter Server 训练中使用，用来启动 worker 端服务
- stop_worker()
  - 训练结束后，停止 worker
- distributed_optimizer(optimizer, strategy=None)
  - 分布式优化算法装饰器，用户可带入单机 optimizer，并配置分布式训练策略，返回一个分布式的 optimizer

RoleMaker

- PaddleCloudRoleMaker
  - 描述：PaddleCloudRoleMaker 是一个高级封装，支持使用 paddle.distributed.launch 或者 paddle.distributed.launch_ps 启动脚本
  - Parameter Server 训练示例：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet
```

(下页继续)

(续上页)

```
role = role_maker.PaddleCloudRoleMaker()
fleet.init(role)
```

- 启动方法：

```
python -m paddle.distributed.launch_ps --worker_num 2 --server_num 2 trainer.
↪py
```

- Collective 训练示例：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

role = role_maker.PaddleCloudRoleMaker(is_collective=True)
fleet.init(role)
```

- 启动方法：

```
python -m paddle.distributed.launch trainer.py
```

- UserDefinedRoleMaker
  - 描述：用户自定义节点的角色信息，IP 和端口信息
  - 示例：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

role = role_maker.UserDefinedRoleMaker(
    current_id=0,
    role=role_maker.Role.SERVER,
    worker_num=2,
    server_endpoints=["127.0.0.1:36011", "127.0.0.1:36012"])

fleet.init(role)
```

CHAPTER 7

昆仑 XPU 芯片运行飞桨

百度昆仑 AI 计算处理器（Baidu KUNLUN AI Computing Processor）是百度集十年 AI 产业技术实践于 2019 年推出的全功能 AI 芯片。基于自主研发的先进 XPU 架构，为云端和边缘端的人工智能业务而设计。百度昆仑与飞桨及其他国产软硬件强强组合，打造一个全面领先的国产化 AI 技术生态，部署和应用于诸多“人工智能 + “的行业领域，包括智能云和高性能计算，智慧制造、智慧城市和安防等。更多昆仑 XPU 芯片详情及技术指标请 点击这里 。参考以下内容可快速了解和体验昆仑 XPU 芯片上运行飞桨：

- 飞桨对昆仑 XPU 芯片的支持：飞桨支持昆仑 XPU 芯片运行
- 飞桨框架昆仑 xpu 版安装说明：飞桨框架昆仑 xpu 版安装说明
- 飞桨框架昆仑 XPU 版训练示例：飞桨框架昆仑 XPU 版训练示例

7.1 飞桨对昆仑 XPU 芯片的支持

自飞桨 2.0 版本起支持昆仑 XPU，目前基于昆仑 XPU 和 X86（Intel）CPU 可实现 12 个模型单机单卡/单机多卡的训练，如下图所示：

模型放置在飞桨模型套件中，各领域套件是 github.com/PaddlePaddle 下的独立 repo，clone 下载即可获取所需的模型文件：

随着 ARM 架构的高性能、低功耗、低成本的优势日益突显，ARM CPU 更多地进入 PC 和服务器领域，众多新锐国产 CPU 也纷纷采用 ARM 架构。在这一趋势下，我们开始尝试在飞腾 CPU 和昆仑 XPU 上运行飞桨，当前已验证 ResNet50 的训练效果。

更多的常用模型以及动态图组网将在后续版本增加。高性能预测库 PaddleInference、PaddleLite、PaddleServing 将在近期发布的新版本中支持昆仑 XPU。敬请期待。

7.2 飞桨框架昆仑 XPU 版安装说明

飞桨提供两种安装方式：

1. 预编译的支持昆仑 XPU 的 wheel 包

目前此 wheel 包只支持两种环境：

英特尔 CPU+ 昆仑 XPU+Ubuntu 系统

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

2. 源码编译安装

其他环境请选择源码编译安装。

7.2.1 安装方式一：通过 Wheel 包安装

下载安装包

环境 1：英特尔 CPU+ 昆仑 XPU+Ubuntu 系统

Python3.7

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.0.0-cp37-cp37m-linux_
↪x86_64.whl
```

```
python3.7 -m pip install -U paddlepaddle-2.0.0-cp37-cp37m-linux_x86_64.whl
```

Python3.6

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.0.0-cp36-cp36m-linux_
↪x86_64.whl
```

```
python3.6 -m pip install -U ``paddlepaddle-2.0.0-cp36-cp36m-linux_x86_64.whl
```

Python2.7

```
Wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.0.0-cp27-cp27mu-linux_
↪x86_64.whl
```

```
python2.7 -m pip install -U ``paddlepaddle-2.0.0-cp27-cp27m-linux_x86_64.whl
```

环境 2：飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

Python3.7

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.0.0-cp37-cp37m-linux_
↪aarch64.whl
```

```
python3.7 -m pip install -U paddlepaddle-2.0.0-cp37-cp37m-linux_aarch64.whl
```

Python3.6

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.0.0-cp36-cp36m-linux_
↪aarch64.whl
```

```
python3.6 -m pip install -U paddlepaddle-2.0.0-cp36-cp36m-linux_aarch64.whl
```

如果使用预编译的支持昆仑 XPU 的 wheel 包出现环境问题，推荐使用源码自行编译支持昆仑 XPU 的包。

验证安装安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

7.2.2 安装方式二：从源码编译支持昆仑 XPU 的包

环境准备

英特尔 CPU+ 昆仑 XPU+Ubuntu 系统

- **处理器：Intel(R) Xeon(R) Gold 6148 CPU @2.40GHz**
- **操作系统：Ubuntu 16.04.6 LTS**
- **Python 版本：2.7/3.6/3.7 (64 bit)**
- **pip 或 pip3 版本：9.0.1+ (64 bit)**
- **cmake 版本：3.10+**
- **gcc/g++ 版本：8.2+**

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

- **处理器：Phytium,FT-2000+/64**
- **操作系统：Kylin release V10 (SP1)/(Tercel)-aarch64-Build04/20200711**
- **Python 版本：3.6/3.7 (64 bit)**

- **pip 或 pip3 版本：9.0.1+ (64 bit)**
- **cmake 版本：3.10+**
- **gcc/g++ 版本：8.2+**

源码编译安装步骤：

1. Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 >=3.10，如果操作系统提供的源包括了合适版本的 cmake，直接安装即可，否则需要

```
wget https://github.com/Kitware/CMake/releases/download/v3.16.8/cmake-3.16.8.tar.gz
tar -xzf cmake-3.16.8.tar.gz && cd cmake-3.16.8
./bootstrap && make && sudo make install
```

1. Paddle 内部使用 patchelf 来修改动态库的 rpath，如果操作系统提供的源包括了 patchelf，直接安装即可，否则需要源码安装，请参考

```
./bootstrap.sh
./configure
make
make check
sudo make install
```

1. 根据requirments.txt安装 Python 依赖库
2. 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle
```

1. 建议切换到 release2.0 分支下进行编译：

```
git checkout [``分支名 ``]
```

例如：

```
git checkout release/2.0
```

1. 并且请创建并进入一个叫 build 的目录下

```
mkdir build && cd build
```

1. 链接过程中打开文件数较多，可能超过系统默认限制导致编译出错，设置进程允许打开的最大文件数：

```
ulimit -n 4096
```

1. 执行 cmake
2. 具体编译选项含义请参见编译选项表

英特尔 CPU+ 昆仑 XPU+Ubuntu 系统

Python3

```
cmake .. -DPY_VERSION=3 -DPYTHON_EXECUTABLE=`which python3` -DWITH_MKL=OFF -DWITH_
↪XPU=ON -DWITH_GPU=OFF -DWITH_TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DWITH_XPU_
↪BKCL=ON
```

Python2

```
cmake .. -DPY_VERSION=2 -DPYTHON_EXECUTABLE=`which python2` -DWITH_MKL=OFF -DWITH_
↪XPU=ON -DWITH_GPU=OFF -DWITH_TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DWITH_XPU_
↪BKCL=ON
```

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

Python3

```
cmake .. -DPY_VERSION=3 -DPYTHON_EXECUTABLE=`which python3` -DWITH_ARM=ON -DWITH_
↪TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_XBYAK=OFF -DWITH_XPU=ON␣
↪-DWITH_GPU=OFF -DWITH_LITE=ON -DLITE_GIT_TAG=develop -DWITH_AARCH64=ON
```

1. 使用以下命令来编译

```
make -j$(nproc)
```

1. 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。
2. 将生成的.whl 包 copy 至带有昆仑 XPU 的目标机器上，并在目标机器上根据requirments.txt安装 Python 依赖库。（如果编译机器同时为带有昆仑 XPU 的目标机器，略过此步）
3. 在带有昆仑 XPU 的目标机器安装编译好的.whl 包：pip install -U（whl 包的名字）或 pip3 install -U（whl 包的名字）。恭喜，至此您已完成昆仑 XPU 机器上 PaddlePaddle 的编译安装。

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

如何卸载

使用以下命令卸载 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

7.3 飞桨框架昆仑 XPU 版训练示例

使用 XPU 训练与 cpu/gpu 相同，只需要加上-o use_xpu=True, 表示执行在昆仑设备上。

7.3.1 ResNet50 下载并运行示例：

模型文件下载命令：

```
cd path_to_clone_PaddleClas
git clone -b release/static https://github.com/PaddlePaddle/PaddleClas.git
```

也可以访问 PaddleClas 的github repo直接下载源码。

运行训练：

```
#FLAGS 指定单卡或多卡训练，此示例运行 2 个卡
export FLAGS_selected_xpus=0,1
# 启动训练
Python3.7 tools/train_multi_platform.py -c configs/kunlun/ResNet50.yaml -o use_
↪gpu=False -o use_xpu=True
```

注意：飞腾 CPU+ 昆仑 XPU 的环境下暂未支持多卡训练。

CHAPTER 8

自定义 OP

本部分将指导您如何新增 Operator，也包括一些必要的注意事项

- 如何写新的 C++ op
- C++ op 相关注意事项
- 如何写新的 Python op
- 如何在框架外部自定义 C++ op

8.1 如何写新的 C++ OP

8.1.1 概念简介

简单介绍需要用到基类，详细介绍请参考设计文档。

- `framework::OperatorBase`: Operator(简写，Op) 基类。
- `framework::OpKernel`: Op 计算函数的基类，称作 Kernel。
- `framework::OperatorWithKernel`：继承自 OperatorBase，Op 有计算函数，称作有 Kernel。
- `framework::OpProtoAndCheckerMaker`：描述该 Op 的输入、输出、属性、注释，主要用于 Python API 接口生成。

根据是否包含 Kernel，可以将 Op 分为两种：包含 Kernel 的 Op 和不包含 kernel 的 Op：

- 包含 Kernel 的 Op 继承自 `OperatorWithKernel`，这类 Op 的功能实现与输入的数据类型、数据布局、数据所在的设备以及 Op 实现所调用第三方库等有关。比如 ConvOp，如果使用 CPU 计算，一般通过调用 mkl 库中的矩阵乘操作实现，如果使用 GPU 计算，一般通过调用 cublas 库中的矩阵乘操作实现，或者直接调用 cudnn 库中的卷积操作。
- 不包含 Kernel 的 Op 继承自 `OperatorBase`，因为这类 Op 的功能实现与设备以及输入的数据不相关。比如 WhileOp、IfElseOp 等。

本教程主要介绍带 Kernel 的 Op 如何写，简单总结 Op 需要包含的内容如下：

实现新的 op 都添加至目录paddle/fluid/operators下，文件命名以 `*_op.h`（如有）、`*_op.cc`、`*_op.cu`（如有）结尾。**系统会根据文件名自动构建 op 和其对应的 Python 扩展**。

下面以矩阵乘操作，即MulOp为例来介绍如何写带 Kernel 的 Operator。

8.1.2 实现 C++ 类

定义 ProtoMaker 类

矩阵乘法的公式：$Out = X * Y$, 可见该计算由两个输入，一个输出组成。

首先定义 `ProtoMaker` 来描述该 Op 的输入、输出，并添加注释：

```
class MulOpMaker : public framework::OpProtoAndCheckerMaker {
 public:
  void Make() override {
    AddInput("X", "(Tensor), The first input tensor of mul op.");
    AddInput("Y", "(Tensor), The second input tensor of mul op.");
    AddOutput("Out", "(Tensor), The output tensor of mul op.");
    AddAttr<bool>("use_mkldnn",
                  "(bool, default false) Only used in mkldnn kernel")
        .SetDefault(false);
    AddAttr<int>(
        "x_num_col_dims",
        R"DOC((int, default 1), The mul_op can take tensors with more than two
              dimensions as its inputs. If the input $X$ is a tensor with more
              than two dimensions, $X$ will be flattened into a two-dimensional
              matrix first. The flattening rule is: the first `num_col_dims`
              will be flattened to form the first dimension of the final matrix
              (the height of the matrix), and the rest `rank(X) - num_col_dims`
              dimensions are flattened to form the second dimension of the final
              matrix (the width of the matrix). As a result, height of the
              flattened matrix is equal to the product of $X$'s first
              `x_num_col_dims` dimensions' sizes, and width of the flattened
              matrix is equal to the product of $X$'s last `rank(x) - num_col_dims`
              dimensions' size. For example, suppose $X$ is a 6-dimensional
```

(下页继续)

(续上页)

```
              tensor with the shape [2, 3, 4, 5, 6], and `x_num_col_dims` = 3.
              Thus, the flattened matrix will have a shape [2 x 3 x 4, 5 x 6] =
              [24, 30].
        )DOC")
        .SetDefault(1)
        .EqualGreaterThan(1);
    AddAttr<int>(
        "y_num_col_dims",
        R"DOC((int, default 1), The mul_op can take tensors with more than two,
              dimensions as its inputs. If the input $Y$ is a tensor with more
              than two dimensions, $Y$ will be flattened into a two-dimensional
              matrix first. The attribute `y_num_col_dims` determines how $Y$ is
              flattened. See comments of `x_num_col_dims` for more details.
        )DOC")
        .SetDefault(1)
        .EqualGreaterThan(1);
    AddAttr<float>(
        "scale_x",
        "scale_x to be used for int8 mul input data x. scale_x has the"
        "same purpose as scale_in in OPs that support quantization."
        "Only to be used with MKL-DNN INT8")
        .SetDefault(1.0f);
    AddAttr<std::vector<float>>(
        "scale_y",
        "scale_y to be used for int8 mul input data y. scale_y has the"
        "same purpose as scale_weights in OPs that support quantization."
        "Only to be used with MKL-DNN INT8")
        .SetDefault({1.0f});
    AddAttr<float>("scale_out",
                   "scale_out to be used for int8 output data."
                   "Only used with MKL-DNN INT8")
        .SetDefault(1.0f);
    AddAttr<bool>(
        "force_fp32_output",
        "(bool, default false) Force quantize kernel output FP32, only "
        "used in quantized MKL-DNN.")
        .SetDefault(false);
    AddComment(R"DOC(
Mul Operator.
This operator is used to perform matrix multiplication for input $X$ and $Y$.
The equation is:
$$Out = X * Y$$
Both the input $X$ and $Y$ can carry the LoD (Level of Details) information,
```

(下页继续)

(续上页)

```
or not. But the output only shares the LoD information with input $X$.
)DOC");
  }
};
```

MulOpMaker继承自 framework::OpProtoAndCheckerMaker。

开发者通过覆盖 framework::OpProtoAndCheckerMaker 中的 Make 函数来定义 Op 所对应的 Proto，通过 AddInput 添加输入参数，通过 AddOutput 添加输出参数，通过 AddAttr 添加属性参数，通过 AddComment 添加 Op 的注释。这些函数会将对应内容添加到 OpProto 中。

上面的代码在 MulOp 中添加两个输入 X 和 Y，添加了一个输出 Out，以及 use_mkldnn 等属性，并解释了各自含义，命名请遵守命名规范。

定义 GradOpMaker 类

通常情况下，大部分 Op 只有一个对应的反向 Op，每个 Op 的会有一个对应的 GradOpMaker。为方便代码编写，paddle 为只有一个反向的 Op 提供了一个模板类SingleGradOpMaker。MulOp 的 GradOpMaker 需要继承这个模板类，并在 Apply() 方法中设置反向 Op 的输入、输出和属性。此外，paddle 还提供了一个默认的 GradOpMaker，DefaultGradOpMaker，该模板类会使用前向 Op 的全部输入 (Input) 输出 (Output) 以及输出变量所对应的梯度（Output@Grad）作为反向 Op 的输入，将前向 Op 的输入变量所对应的的梯度（Input@Grad）作为输出。

注意: 不要将反向 Op 不会用到的变量放到反向 Op 的输入列表中，这样会导致这些不会被反向 Op 用到的变量的空间不能够及时回收，进而有可能导致用到该 Op 的模型可以设置的 batch_size 较低。比如 relu 操作的前向操作为: out.device(d) = x.cwiseMax(static_cast<T>(0)); 反向操作为: dx.device(d) = dout * (out > static_cast<T>(0)).template cast<T>();。显然，反向操作中只是用到了 out、dout、dx，没有用到 x。因此，通常不建议使用默认的 DefaultGradOpMaker。

下面示例定义了 MulOp 的 GradOpMaker。

```
template <typename T>
class MulOpGradMaker : public framework::SingleGradOpMaker<T> {
 public:
  using framework::SingleGradOpMaker<T>::SingleGradOpMaker;

 protected:
  void Apply(GradOpPtr<T> retv) const override {
    retv->SetType("mul_grad");
    retv->SetInput("X", this->Input("X"));
    retv->SetInput("Y", this->Input("Y"));
    retv->SetInput(framework::GradVarName("Out"), this->OutputGrad("Out"));
    retv->SetOutput(framework::GradVarName("X"), this->InputGrad("X"));
    retv->SetOutput(framework::GradVarName("Y"), this->InputGrad("Y"));
```

(下页继续)

(续上页)

```
    retv->SetAttrMap(this->Attrs());
  }
};
```

注意：

- 有些 Op 的前向逻辑和反向逻辑是一样的，比如`ScaleOp`. 这种情况下，前向 Op 和反向 Op 的 Kernel 可以为同一个。
- 有些前向 Op 所对应的反向 Op 可能有多个，比如`SumOp`，这种情况下，`GradMaker` 需要继承 `framework::GradOpDescMakerBase`。
- 有些 Op 的反向对应另一个 Op 的前向，比如`SplitOp`，这种情况下，`SplitGradMaker`中定义的 `SplitOp` 反向 Op 的 Type 就是 `concat`，
- 为高效地同时支持命令式编程模式 (动态图) 和声明式编程模式 (静态图)，`SingleGradOpMaker` 是一个模板类，在注册 Operator 时需要同时注册 `MulOpGradMaker<OpDesc>`（声明式编程模式使用）和 `MulOpGradMaker<OpBase>`（命令式编程模式使用）。

定义 Operator 类

下面实现了 MulOp 的定义：

```
class MulOp : public framework::OperatorWithKernel {
 public:
  using framework::OperatorWithKernel::OperatorWithKernel;

  void InferShape(framework::InferShapeContext* ctx) const override {
    PADDLE_ENFORCE_EQ(
        ctx->HasInput("X"), true,
        platform::errors::NotFound("Input(X) of MulOp should not be null."));
    PADDLE_ENFORCE_EQ(
        ctx->HasInput("Y"), true,
        platform::errors::NotFound("Input(Y) of MulOp should not be null."));
    PADDLE_ENFORCE_EQ(
        ctx->HasOutput("Out"), true,
        platform::errors::NotFound("Output(Out) of MulOp should not be null."));

    auto x_dims = ctx->GetInputDim("X");
    auto y_dims = ctx->GetInputDim("Y");

    int x_num_col_dims = ctx->Attrs().Get<int>("x_num_col_dims");
    int y_num_col_dims = ctx->Attrs().Get<int>("y_num_col_dims");

    VLOG(3) << "mul operator x.shape=" << x_dims << " y.shape=" << y_dims
```

(下页继续)

(续上页)

```
          << " x_num_col_dims=" << x_num_col_dims
          << " y_num_col_dims=" << y_num_col_dims;

  PADDLE_ENFORCE_NE(framework::product(y_dims), 0,
                    platform::errors::PreconditionNotMet(
                        "The Input variable Y(%s) has not "
                        "been initialized. You may need to confirm "
                        "if you put exe.run(startup_program) "
                        "after optimizer.minimize function.",
                        ctx->Inputs("Y").front()));
  PADDLE_ENFORCE_GT(
      x_dims.size(), x_num_col_dims,
      platform::errors::InvalidArgument(
          "The input tensor X's dimensions of MulOp "
          "should be larger than x_num_col_dims. But received X's "
          "dimensions = %d, X's shape = [%s], x_num_col_dims = %d.",
          x_dims.size(), x_dims, x_num_col_dims));
  PADDLE_ENFORCE_GT(
      y_dims.size(), y_num_col_dims,
      platform::errors::InvalidArgument(
          "The input tensor Y's dimensions of MulOp "
          "should be larger than y_num_col_dims. But received Y's "
          "dimensions = %d, Y's shape = [%s], y_num_col_dims = %d.",
          y_dims.size(), y_dims, y_num_col_dims));

  auto x_mat_dims = framework::flatten_to_2d(x_dims, x_num_col_dims);
  auto y_mat_dims = framework::flatten_to_2d(y_dims, y_num_col_dims);

  PADDLE_ENFORCE_EQ(
      x_mat_dims[1], y_mat_dims[0],
      platform::errors::InvalidArgument(
          "After flatten the input tensor X and Y to 2-D dimensions "
          "matrix X1 and Y1, the matrix X1's width must be equal with matrix "
          "Y1's height. But received X's shape = [%s], X1's shape = [%s], "
          "X1's "
          "width = %s; Y's shape = [%s], Y1's shape = [%s], Y1's height = "
          "%s.",
          x_dims, x_mat_dims, x_mat_dims[1], y_dims, y_mat_dims,
          y_mat_dims[0]));
  std::vector<int64_t> output_dims;
  output_dims.reserve(
      static_cast<size_t>(x_num_col_dims + y_dims.size() - y_num_col_dims));
```

(下页继续)

(续上页)

```
    for (int i = 0; i < x_num_col_dims; ++i) {
      output_dims.push_back(x_dims[i]);
    }

    for (int i = y_num_col_dims; i < y_dims.size(); ++i) {
      output_dims.push_back(y_dims[i]);
    }

    ctx->SetOutputDim("Out", framework::make_ddim(output_dims));
    ctx->ShareLoD("X", /*->*/ "Out");
  }

  framework::OpKernelType GetExpectedKernelType(
      const framework::ExecutionContext& ctx) const {
    framework::LibraryType library = framework::LibraryType::kPlain;
    framework::DataLayout layout = framework::DataLayout::kAnyLayout;
    int customized_type_value =
        framework::OpKernelType::kDefaultCustomizedTypeValue;
    auto input_data_type = OperatorWithKernel::IndicateVarDataType(ctx, "X");
#ifdef PADDLE_WITH_MKLDNN
    if (library == framework::LibraryType::kPlain &&
        platform::CanMKLDNNBeUsed(ctx)) {
      library = framework::LibraryType::kMKLDNN;
      layout = framework::DataLayout::kMKLDNN;

      if (input_data_type == framework::DataTypeTrait<int8_t>::DataType() ||
          input_data_type == framework::DataTypeTrait<uint8_t>::DataType()) {
        customized_type_value = kMULMKLDNNINT8;
      }
    }
#endif

    return framework::OpKernelType(input_data_type, ctx.GetPlace(), layout,
                                   library, customized_type_value);
  }
};
```

MulOp继承自 OperatorWithKernel。public 成员：

```
using framework::OperatorWithKernel::OperatorWithKernel;
```

这句表示使用基类 OperatorWithKernel 的构造函数，也可写成：

```
MulOp(const std::string &type, const framework::VariableNameMap &inputs,
      const framework::VariableNameMap &outputs,
      const framework::AttributeMap &attrs)
  : OperatorWithKernel(type, inputs, outputs, attrs) {}
```

此外，Operator 类通常需要重写 `InferShape` 接口，并在有必要时重写 `GetExpectedKernelType` 接口。`InferShape` 为 const 函数，不能修改 Op 的成员变量，参数为 `framework::InferShapeContext* ctx`，通过该参数可获取到输入输出以及属性。它的功能是：

- 做检查，尽早报错：检查输入数据维度、类型等是否合法。
- 设置输出 Tensor 的形状以及 LoD 信息。

`GetExpectedKernelType` 接口 OperatorWithKernel 类中用于获取指定设备（例如 CPU，GPU）上指定数据类型（例如 double，float）的 OpKernel 的方法。该方法的重写可见请参考写 C++ OP 相关注意事项。

通常 `OpProtoMaker` 和 `Op` 类的定义写在`.cc` 文件中，和下面将要介绍的注册函数一起放在`.cc` 中

InferShape 区分 compile time 和 run time

在我们的声明式编程模式网络中，`InferShape` 操作在编译时 (compile time) 和运行时 (run time)都会被调用，在 compile time 时，由于真实的维度未知，框架内部用-1 来表示，在 run time 时，用实际的维度表示，因此维度的值在 compile time 和 run time 时可能不一致，如果存在维度的判断和运算操作，InferShape 就需要区分 compile time 和 run time。

以下两种情况需要区分 compile time 和 run time。

1. 检查

如以下代码：

```
auto x_dim = ctx->GetInputDim("X");
int i = xxx;
PADDLE_ENFORCE_GT( x_dim[i] , 10)
```

在 compile time 的时候，x_dim[i] 可能等于-1，导致这个 PADDLE_ENFORCE_GT 报错退出。

如果用了以下 paddle 中定义的宏进行判断：

```
PADDLE_ENFORCE_EQ ( x_dim[i] , 10)
PADDLE_ENFORCE_NE ( x_dim[i] , 10)
PADDLE_ENFORCE_GT ( x_dim[i] , 10)
PADDLE_ENFORCE_GE ( x_dim[i] , 10)
PADDLE_ENFORCE_LT ( x_dim[i] , 10)
PADDLE_ENFORCE_LE ( x_dim[i] , 10)
```

都需要区分 compile time 和 run time

2. 运算

如以下代码:

```
auto x_dim = ctx->GetInputDim("X");
int i = xxx;
y_dim[0] = x_dim[i] + 10
```

在 compile time 的时候，x_dim[i] 可能等于-1，得到的 y_dim[0] 等于 9，是不符合逻辑的

如果用到了类似以下的运算操作

```
y_dim[i] = x_dim[i] + 10
y_dim[i] = x_dim[i] - 10
y_dim[i] = x_dim[i] * 10
y_dim[i] = x_dim[i] / 10
y_dim[i] = x_dim[i] + z_dim[i]
```

都需要区分 compile time 和 run time

处理的标准:

- 检查：compile time 的时候不判断维度等于-1 的情况，但在 runtime 的时候检查
- 运算：-1 和其他数做任何运算都要等于-1

参考代码

1. 判断的实现方法可以参考cross_entropy_op，cross_entropy_op 要求 X 和 labels 的两个输入，除了最后一维以外，其他的维度完全一致

```
  bool contain_unknown_dim = framework::contain_unknown_dim(x_dims) ||
                             framework::contain_unknown_dim(label_dims);
  bool check = ctx->IsRuntime() || !contain_unknown_dim;
  if (check) {
    PADDLE_ENFORCE_EQ(framework::slice_ddim(x_dims, 0, rank - 1),
                      framework::slice_ddim(label_dims, 0, rank - 1),
                      "Input(X) and Input(Label) shall have the same shape "
                      "except the last dimension.");
  }
```

1. 运算的实现可以参考concat_op，concat 在 InferShape 判断时，调用 `ComputeAndCheckShape`，除了进行 concat 轴之外，其他的维度完全一致；在生成 output 的维度时，把 concat 轴的维度求和，其他的维度和输入保持一致。

```
  const size_t n = inputs_dims.size();
  auto out_dims = inputs_dims[0];
  size_t in_zero_dims_size = out_dims.size();
```

(下页继续)

(续上页)

```
    for (size_t i = 1; i < n; i++) {
      for (size_t j = 0; j < in_zero_dims_size; j++) {
        if (j == axis) {
          if (is_runtime) {
            out_dims[axis] += inputs_dims[i][j];
          } else {
            if (inputs_dims[i][j] == -1) {
              out_dims[axis] = -1;
            } else {
              out_dims[axis] += inputs_dims[i][j];
            }
          }
        } else {
          bool check_shape =
              is_runtime || (out_dims[j] > 0 && inputs_dims[i][j] > 0);
          if (check_shape) {
            // check all shape in run time
            PADDLE_ENFORCE_EQ(
                inputs_dims[0][j], inputs_dims[i][j],
                "ShapeError: Dimension %d in inputs' shapes must be equal. "
                "But recevied input[0]'s shape = "
                "[%s], input[%d]'s shape = [%s].",
                j, inputs_dims[0], i, inputs_dims[i]);
          }
        }
      }
    }
```

定义 OpKernel 类

MulKernel 继承自 framework::OpKernel，带有下面两个模板参数:

- typename DeviceContext: 表示设备类型。不同设备 (CPU、CUDA) 共享同一个 Kernel 时，需加该模板参数；不共享则不加，一个不共享的例子是SGDOpKernel。
- typename T : 表示数据类型，如 float, double, int16 等。

需要为 MulKernel 类重写 Compute 接口。

- Compute 接受一个输入参数: const framework::ExecutionContext& context。
- 与 InferShapeContext 相比，ExecutionContext 增加了设备类型，同样可获取到输入输出和属性参数。
- Compute 函数里实现 OpKernel 的具体计算逻辑。

Op 的输入和输出可分别通过 ExecutionContext::Input<T>() 和 ExecutionContext::Output<T>() 获得。

注意： 若 op 的输入/输出的变量类型是 LoDTensor（paddle 默认所有的 Tensor 默认都是 LoDTensor 类型），请写成 ExecutionContext::Input<LoDTensor>() 和 ExecutionContext::Output<LoDTensor>()，不要写 ExecutionContext::Input<Tensor>() 和 ExecutionContext::Output<Tensor>()。因为若实际的变量类型为 SelectedRows，Input<Tensor>() 和 Output<Tensor>() 方法会将 SelectedRows 类型特化为 Tensor，导致潜在的错误。

下面是 MulKernel Compute 的实现：

```
template <typename DeviceContext, typename T>
class MulKernel : public framework::OpKernel<T> {
 public:
  void Compute(const framework::ExecutionContext& context) const override {
    const Tensor* x = context.Input<Tensor>("X");
    const Tensor* y = context.Input<Tensor>("Y");
    Tensor* z = context.Output<Tensor>("Out");
    const Tensor x_matrix =
        x->dims().size() > 2
            ? framework::ReshapeToMatrix(
                  *x, context.template Attr<int>("x_num_col_dims"))
            : *x;
    const Tensor y_matrix =
        y->dims().size() > 2
            ? framework::ReshapeToMatrix(
                  *y, context.template Attr<int>("y_num_col_dims"))
            : *y;

    z->mutable_data<T>(context.GetPlace());
    auto z_dim = z->dims();
    if (z_dim.size() != 2) {
      z->Resize({x_matrix.dims()[0], y_matrix.dims()[1]});
    }

    auto blas = math::GetBlas<DeviceContext, T>(context);

    blas.MatMul(x_matrix, y_matrix, z);
    if (z_dim.size() != 2) {
      z->Resize(z_dim);
    }
  }
};
```

需要注意：**不同设备 (CPU、CUDA) 共享一个 Op 定义，是否则共享同一个 OpKernel，取决于 Compute**

调用的函数是否支持不同设备。

MulOp 的 CPU、CUDA 实现共享同一个 Kernel。OpKernel 不共享的例子可以参考：SGDOpKernel。

为了使 OpKernel 的计算过程书写更加简单，并且 CPU、CUDA 的代码可以复用，我们通常借助 Eigen unsupported Tensor 模块来实现 Compute 接口。关于在 PaddlePaddle 中如何使用 Eigen 库，请参考使用文档。

到此，前向 Op 实现完成。接下来，需要在.cc 文件中注册该 op 和 kernel。反向 Op 类的定义，反向 OpKernel 的定义与前向 Op 类似，这里不再赘述。

注册 Operator

- 在.cc 文件中注册前向、反向 Op 类，注册 CPU Kernel。

```
namespace ops = paddle::operators;
REGISTER_OPERATOR(mul, ops::MulOp, ops::MulOpMaker, ops::MulOpInferVarType,
                  ops::MulOpGradMaker<paddle::framework::OpDesc>,
                  ops::MulOpGradMaker<paddle::imperative::OpBase>);

REGISTER_OPERATOR(mul_grad, ops::MulGradOp);

REGISTER_OP_CPU_KERNEL(mul,
              ops::MulKernel<paddle::platform::CPUDeviceContext, float>,
              ops::MulKernel<paddle::platform::CPUDeviceContext, double>);
REGISTER_OP_CPU_KERNEL(mul_grad,
              ops::MulGradKernel<paddle::platform::CPUDeviceContext, float>,
              ops::MulGradKernel<paddle::platform::CPUDeviceContext, double>);
```

在上面的代码中，使用 REGISTER_OPERATOR 注册了 ops::MulOp 类，类型名为 mul，该类的 ProtoMaker 为 ops::MulOpMaker，其 GradOpMaker 分别是 ops::MulOpGradMaker<paddle::framework::OpDesc>(声明式编程模式使用) 和 ops::MulOpGradMaker<paddle::imperative::OpBase>(命令式编程模式使用)，并使用 REGISTER_OPERATOR 注册 ops::MulGradOp，类型名为 mul_grad。然后，使用 REGISTER_OP_CPU_KERNEL 注册了 ops::MulKernel 类，并特化模板参数为设备为 paddle::platform::CPUPlace、数据类型为 float 类型和 double 类型；同理，注册 ops::MulGradKernel 类。

- 在 .cu 文件中注册 CUDA Kernel。
  - 请注意，如果 CUDA Kernel 的实现基于 Eigen unsupported 模块，那么在 .cu 的开始请加上宏定义 #define EIGEN_USE_GPU，代码示例如下：

```
// if use Eigen unsupported module before include head files
#define EIGEN_USE_GPU

namespace ops = paddle::operators;
```

(下页继续)

(续上页)

```
REGISTER_OP_CUDA_KERNEL(mul,
                        ops::MulKernel<paddle::platform::CUDADeviceContext, float>
↪,
                        ops::MulKernel<paddle::platform::CUDADeviceContext,␣
↪double>);
REGISTER_OP_CUDA_KERNEL(mul_grad,
                        ops::MulGradKernel<paddle::platform::CUDADeviceContext,␣
↪float>,
                        ops::MulGradKernel<paddle::platform::CUDADeviceContext,␣
↪double>);
```

注意：

在运行 Op 时，框架系统会根据输入数据所在的设备、输入数据的类型等信息自动的选择合适的 OpKernel，比如输入的数据是在 GPU 上，并且为 `float` 类型，框架系统会选择由 `REGISTER_OP_CUDA_KERNEL` 注册的 `ops::MulKernel<paddle::platform::CUDADeviceContext, float>`。如果用户希望指定运行时可被调用的 OpKernel，用户需要覆盖 `framework::OperatorWithKernel` 中的 `GetExpectedKernelType` 函数，比如 `MulOp` 会根据属性 `use_mkldnn` 为 `false` 还是为 `true` 决定是否调用 mkldnn 库来完成计算。

编译

详细的编译环境准备和执行流程可参考从源码编译，下面简单介绍几个主要步骤。在 `Paddle` 代码目录下创建并切换到 build 目录：

```
mkdir build && cd build
```

执行 `cmake` 命令，具体选项可参考从源码编译中的介绍，下面的命令为编译 Python3.5，GPU 版本，带测试，Release 版本的 Paddle。

```
cmake .. -DPY_VERSION=3.5 -DWITH_GPU=ON -DWITH_TESTING=ON -DCMAKE_BUILD_TYPE=Release
```

在 `build` 目录下，运行下面命令可以进行编译整个 paddle：

```
make -j$(nproc)
```

注意： 新增 op 后请重新执行 `cmake` 命令，然后再执行 `make` 命令编译 paddle。

8.1.3 绑定 Python

系统会对新增的 op 自动绑定 Python，并链接到生成的 lib 库中。

使用 mul 操作在 Python 端构建 Layer

在 Python 端，`mul` 操作用于构建 FC 层，即：

$$Out = Act({X*W + b})$$

具体实现方式可参考FC 层的实现代码。

8.1.4 实现单元测试

单测包括对比前向 Op 不同设备 (CPU、CUDA) 的实现、对比反向 OP 不同设备 (CPU、CUDA) 的实现、反向 Op 的梯度测试。下面介绍介绍`MulOp` 的单元测试。

注意：

单测中的测试用例需要尽可能的覆盖 Op 中的所有分支。

前向 Operator 单测

Op 单元测试继承自 `OpTest`。各项具体的单元测试在 `TestMulOp` 里完成。测试 Operator，需要：

1. 在 `setUp` 函数定义输入、输出，以及相关的属性参数。

   注意：输入输出请以 `ndarray` 的类型配置输入/输出，如果需要配置一个带 `LOD` 的输入/输出，请以 `tuple` 的形式传入，`tuple` 中应该有两个类型为 `ndarray` 的元素，第一个是实际的数据，第二个是 `LOD`

2. 生成随机的输入数据。
3. 在 Python 脚本中实现与前向 operator 相同的计算逻辑，得到输出值，与 operator 前向计算的输出进行对比。
4. 反向计算已经自动集成进测试框架，直接调用相应接口即可。

```
import unittest
import numpy as np
from op_test import OpTest


class TestMulOp(OpTest):
    def setUp(self):
        self.op_type = "mul"
        self.inputs = {
```

(下页继续)

(续上页)

```
            'X': np.random.random((32, 84)).astype("float32"),
            'Y': np.random.random((84, 100)).astype("float32")
        }
        self.outputs = {'Out': np.dot(self.inputs['X'], self.inputs['Y'])}

    def test_check_output(self):
        self.check_output()

    def test_check_grad_normal(self):
        self.check_grad(['X', 'Y'], 'Out', max_relative_error=0.5)

    def test_check_grad_ingore_x(self):
        self.check_grad(
            ['Y'], 'Out', max_relative_error=0.5, no_grad_set=set("X"))

    def test_check_grad_ingore_y(self):
        self.check_grad(
            ['X'], 'Out', max_relative_error=0.5, no_grad_set=set('Y'))
```

上面的代码首先导入依赖的包，下面是对 `setUp` 函数中操作的重要变量的详细解释：

- `self.op_type = "mul"`：定义类型，与 operator 注册时注册的类型一致。
- `self.inputs`：定义输入，类型为 `numpy.array`，并初始化。
- `self.outputs`：定义输出，并在 Python 脚本中完成与 operator 同样的计算逻辑，返回 Python 端的计算结果。

反向 operator 单测

而反向测试中：

- `test_check_grad_normal` 中调用 `check_grad` 使用数值法检测梯度正确性和稳定性。
  - 第一个参数 `["X", "Y"]`：指定对输入变量 `X`、`Y` 做梯度检测。
  - 第二个参数`"Out"`：指定前向网络最终的输出目标变量 `Out`。
  - 第三个参数 `max_relative_error`：指定检测梯度时能容忍的最大错误值。
- `test_check_grad_ingore_x` 和 `test_check_grad_ingore_y` 分支用来测试只需要计算一个输入梯度的情况。

编译和执行

python/paddle/fluid/tests/unittests/ 目录下新增的 test_*.py 单元测试会被自动加入工程进行编译。

请注意，**运行单元测试测时需要编译整个工程**，并且编译时需要打开 WITH_TESTING。

参考上述编译过程，编译成功后，在 build 目录下执行下面的命令来运行单元测试:

```
make test ARGS="-R test_mul_op -V"
```

或者执行:

```
ctest -R test_mul_op
```

8.1.5 注意事项

- 注册 Op 时的类型名，需要和该 Op 的名字一样。即不允许在 A_op.cc 里面，注册 REGISTER_OPERATOR(B, ...) 等，这将会导致单元测试出错。
- 如果 Op 没有实现 CUDA Kernel，请不要创建空的 *_op.cu，这将会导致单元测试出错。
- 如果多个 Op 依赖一些共用的函数，可以创建非 *_op.* 格式的文件来存放，如 gather.h 文件。

PADDLE_ENFORCE 使用注意

实现 Op 时检查数据的合法性需要使用 PADDLE_ENFORCE 以及 PADDLE_ENFORCE_EQ 等宏定义，基本格式如下:

```
PADDLE_ENFORCE(表达式, 错误提示信息)
PADDLE_ENFORCE_EQ(比较对象 A, 比较对象 B, 错误提示信息)
```

如果表达式为真，或者比较对象 A=B，则检查通过，否则会终止程序运行，向用户反馈相应的错误提示信息。为了确保提示友好易懂，开发者需要注意其使用方法。

总体原则

任何使用了 PADDLE_ENFORCE 与 PADDLE_ENFORCE_XX 检查的地方，必须有详略得当的备注解释！**错误提示信息不能为空！**

提示信息书写标准

1. [required] 哪里错了？为什么错了？
   - 例如：`ValueError: Mismatched label shape`
2. [optional] 期望的输入是什么样的？实际的输入是怎样的？
   - 例如：`Expected labels dimension=1. Received 4.`
3. [optional] 能否给出修改意见？
   - 例 如：`Suggested Fix:If your classifier expects one-hot encoding label, check your n_classes argument to the estimatorand/or the shape of your label.Otherwise, check the shape of your label.`

如果并非必要或者简洁的描述即可表达清楚以上要点，根据情况书写亦可。

FAQ 典型问题

1. 无报错信息或报错信息过于简单，不能给用户提供有效的提示！

   问题示例 1： 未写提示信息

```
PADDLE_ENFORCE(ctx->HasInput("X"), "");
```

   问题示例 2： 提示信息过于简单

```
PADDLE_ENFORCE(i != nullptr, "i must be set"); // i 是什么？
```

2. 在报错信息中使用开发人员定义的变量缩写，不易理解！

   问题示例：

```
PADDLE_ENFORCE(forward_pd != nullptr,
                    "Fail to find eltwise_fwd_pd in device context");  //eltwise_
↪fwd_pd 用户可能看不懂
```

3. OP 内部调用非法接口：Op 内部如果出现 Output = ShareDataWith(Input) 问题示例：

```
auto *out = ctx.Output<framework::LoDTensor>("Out");
auto *in = ctx.Input<framework::LoDTensor>("X");
out->ShareDataWith(*in);
```

   Op 内部如果出现 Output = ShareDataWith(Input)，相当于 operator 图的中有一条隐藏边，连接了 Input 和 Output，这条边无法在图分析中表达，引发基于图优化的错误。

4. OP 实现的性能实践调用了 eigen 的 broadcast, chop 等操作，性能会比手写 cuda kernel 差几倍以上。此时 cpu 的实现可以复用 eigen，gpu 实现可以实现 cuda kernel.

OP InferShape 检查提示信息特别说明

- 检查输入输出变量，请统一遵循以下格式 `Input(变量名) of OP 名 operator should not be null.`

  正确示例：

```
PADDLE_ENFORCE(ctx->HasInput("Input"),
                        "Input(Input) of LSTMP operator should not be null.");
```

- 反向 Op 的输入输出检查，要写明反向 Op 的名字

  正确示例：

```
PADDLE_ENFORCE(ctx->HasInput("X"),
                        "Input(X) of LoDResetGrad opreator should not be null.");
```

8.2 C++ OP 相关注意事项

8.2.1 Paddle 中 Op 的构建逻辑

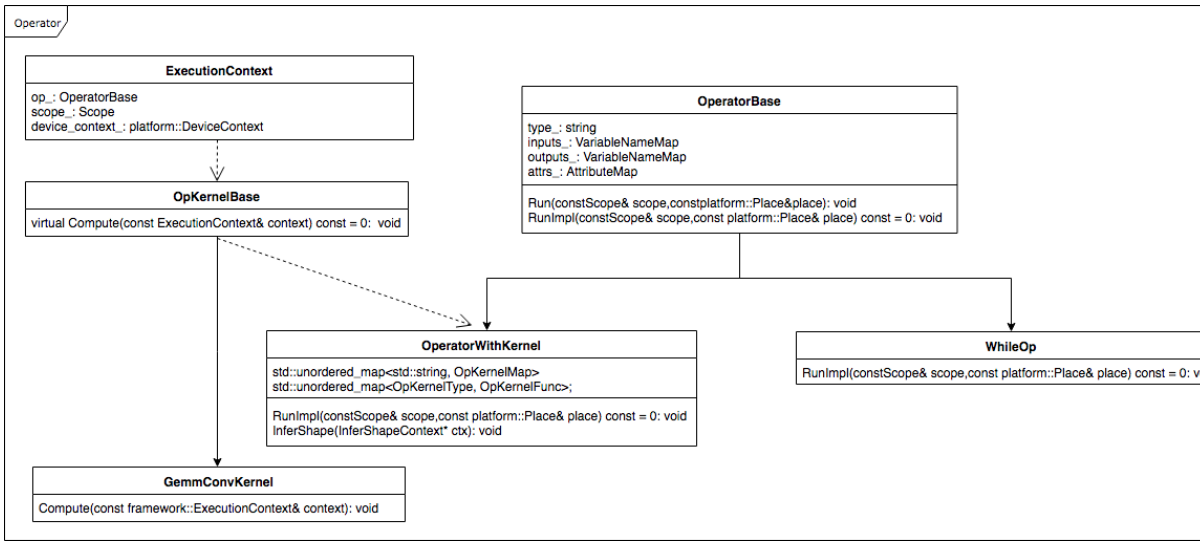
1.Paddle 中 Op 的构建逻辑

Paddle 中所有的 Op 都继承自 `OperatorBase`，且所有的 Op 都是无状态的，每个 Op 包含的成员变量只有四个：type、inputs、outputs、attribute。

Op 的核心方法是 Run，Run 方法需要两方面的资源：数据资源和计算资源，这两个资源分别通过 `Scope` 和 `Place` 获取。框架内部有一个全局的 `DeviceContextPool`，用来记录 `Place` 和 `DeviceContext` 之间的对应的关系，即每个 `Place` 有且仅有一个 `DeviceContext` 与之对应，`DeviceContext` 中存放了当前设备的计算资源。比如对于 GPU，这些资源包括 `cudnn_handle`、`cublas_handle`、`stream` 等，**Op 内部所有的计算（数据拷贝和 CUDA Kernel 等）都必须在 `DeviceContext` 中进行**。

Paddle 框架的设计理念是可以在多种设备及第三方库上运行，有些 Op 的实现可能会因为设备或者第三方库的不同而不同。为此，Paddle 引入了 OpKernel 的方式，即一个 Op 可以有多个 OpKernel，这类 Op 继承自 `OperatorWithKernel`，这类 Op 的代表是 conv_op，conv_op 的 OpKernel 有：`GemmConvKernel`、`CUDNNConvOpKernel`、`ConvMKLDNNOpKernel`，且每个 OpKernel 都有 double 和 float 两种数据类型。不需要 OpKernel 的代表有 `WhileOp` 等。

Operator 继承关系图：



进一步了解可参考：multi_devices，scope，Developer’s_Guide_to_Paddle_Fluid

2.Op 的注册逻辑

每个 Operator 的注册项包括：`C++ OpCreator creator_; GradOpMakerFN grad_op_maker_; proto::OpProto* proto_{nullptr}; OpAttrChecker* checker_{nullptr}; InferVarTypeFN infer_var_type_; InferShapeFN infer_shape_;`

通常 Op 注释时需要调用 REGISTER_OPERATOR，即：`REGISTER_OPERATOR(op_type, OperatorBase op_maker_and_checker_maker, op_grad_opmaker, op_infer_var_shape, op_infer_var_type)`

注意：

1. 对于所有 Op，前三个参数是必须的，op_type 指明 op 的名字，OperatorBase 是该 Op 的对象，op_maker_and_checker_maker 是 op 的 maker 以及 Op 中 attr 的 checker。
2. 如果该 Op 有反向，则必须要有 op_grad_opmaker，因为在 backward 会根据正向的 Op 中获取反向 Op 的 Maker。
3. 框架提供了一个默认的 op_grad_opmaker：`DefaultGradOpDescMaker`，这个 Maker 会将前向 Op 的输入和输出都作为反向 Op 的输入，将前向 Op 的输入的梯度作为反向 Op 的输出，并将前向 Op 的属性拷贝过来。**注意：DefaultGradOpDescMaker 会将前向 Op 的所有输入输出都做反向 Op 的输入，即使这个输入是没有必要的，这将会导致无法对没有用到的变量做内存优化**。
4. 框架没有提供默认的 op_infer_var_shape 方法。如果该 Op 是无 OpKernel 的，通常需要用户添加对应的 op_infer_var_shape 方法；如果该 Op 是有 OpKernel 的，需要实现 `OperatorWithKernel` 中的 `InferShape` 方法，此时不需要提供 op_infer_var_shape 方法。具体实现可参考while_op.cc，conv_op.cc。
5. 框架没有提供默认的 op_infer_var_type 方法，用户需要根据实际情况添加 op_infer_var_type。严格来说每个 Op 都应该注册一个 InferVarType，op_infer_var_type 根据输入的 Var 的 type 和 dtype 推断输出 Var

的 type 和 dtype。**注意：在 Python 端的 LayerHelper 中 create_variable_for_type_inference 操作返回的 Variable 里面是 LoDTensor，C++ 端的 InferVarType 可以修改 `Variable` 的 type 和 dtype**。

更多内容请参考: 如何写新的 Op

8.2.2 写 Op 注意事项

1.Op 可以支持输入输出类型

Paddle 的 Op 的输入输出都是 `Variable`，从设计上讲，`Variable` 中可以存放任意类型，Op 的输入输出 `Variable` 可能是是任意类型，通常情况下 `Variable` 中存放的是 `LoDTensor`、`SelectedRows`。

注意：

- 代码中经常出现 `context.Input<Tensor>("Input")`，并不表示"Input" 的 `Variable` 是 `Tensor`，而是从"Input" 的 `Variable` 的 `LoDTensor` 中获取 `Tensor`。如果"Input" 的 `Variable` 是 `SelectedRows`，则会报错。
- 如果"Input" 是 `SelectedRows`，`context->GetInputDim("Input")` 返回的是 `var->Get<SelectedRows>().GetCompleteDims()`，而不是 `SelectedRows` 中 `Tensor` 的 Dim。

2. 在 Op 内部不能对输入的数据做任何的改写

在 Op 内部绝不允许对输入数据做任何改写，因为可能存在其他 Op 需要读这个数据。

3.OpKernel 需要注册的数据类型

目前要求所有 OpKernel 都要注册 double 和 float 数据类型。

4.GetExpectedKernelType 方法重写

GetExpectedKernelType 方法是 OperatorWithKernel 类中用于获取指定设备（例如 CPU，GPU）上指定数据类型（例如 double，float）的 OpKernel 的方法。该方法通过获取输入变量内部的 Tensor 数据类型得知需要的 Kernel 数据类型，但是由于 Tensor 在此处可能尚未被初始化，所以在该方法内使用输入变量时需要进行必要的初始化检查。在新增含 Kernel 的 Op 的时候，关于该方法的重写需要注意以下两点。

4.1 仅在必要时重写此方法

基类 OperatorWithKernel 中的 GetExpectedKernelType 方法对于派生类 Op 的所有输入变量进行了完备的初始化检查，建议在新增的 Op 中直接使用基类的此方法，例如：

- MeanOp：该 Op 的所有输入变量在 Run 之前应该全部被初始化，初始化检查是必要且合理的

但是在一些情况下，直接使用基类的 GetExpectedKernelType 方法无法满足需求，则需要对该方法进行重写，具体情况及示例如下：

1. OP 的输入有多个，且数据类型不同，例如 AccuracyOp，需要重写 GetExpectedKernelType 方法，指定用某一输入变量获取 kernel 类型
2. Op 包含 Dispensable 的输入变量，该类输入变量是可选的，当用户未输入时，该类变量未被初始化属于合理情况，例如 ConvOp，存在 Bias 等可选的输入变量，需要重写 GetExpectedKernelType 方法，指定用必须提供的输入变量获取 kernel 类型
3. Op 的部分输入变量即使未被初始化也属于合理情况，例如 ConcatOp，输入变量 X 中有个 Tensor 需要连接，其中可能包含未被初始化的 Tensor，需要重写 GetExpectedKernelType 方法，使用输入变量 X 获取 kernel 的过程中，合理忽略掉部分 Tensor 为空的情况
4. OP 的 Kernel 类型与输入变量无关（可能由其他参数指定），例如 FillOp，该 Op 没有输入，Kernel 类型通过 Op 的 dtype 参数指定，因此需要重写 GetExpectedKernelType 方法，用参数指定的数据类型获取 kernel 类型
5. Op Kernel 的部分参数在使用某些库时，需要指定为相应的值，因此需要重写 GetExpectedKernelType 方法，覆盖默认参数
    - 使用 CUDNN 库：需要指定 OpKernel 的 LibraryType 为 kCUDNN，例如 AffineGridOp
    - 使用 MKLDNN 库：需要指定 OpKernel 的 LibraryType 和 DataLayout 为 kMKLDNN MulOp

4.2 重写此方法时需要对输入变量进行初始化检查

在需要重写 GetExpectedKernelType 方法时，一般会根据某一输入变量获取 Kernel 的数据类型，此时请使用 `OperatorWithKernel::IndicateVarDataType` 接口获取变量的 dtype，该方法对指定的输入变量进行了必要的初始化检查，详见Paddle PR #20044，实现示例如下，：

```
framework::OpKernelType GetExpectedKernelType(
    const framework::ExecutionContext& ctx) const override {
  return framework::OpKernelType(
      OperatorWithKernel::IndicateVarDataType(ctx, "X"), ctx.GetPlace());
}
```

如果未使用带有初始化检查的方法，直接使用了 `Tensor->type()`，可能会导致报出 `holder_ should not be null. Tensor not initialized yet when Tensor::type()` 的错误，例如Paddle issue #19522，用户仅凭该错误信息将无法得知具体出错的 Op，不利于调试。

5.Op 兼容性问题

对 Op 的修改需要考虑兼容性问题，要保证 Op 修改之后，之前的模型都能够正常加载及运行，即新版本的 Paddle 预测库能成功加载运行旧版本训练的模型。**所以，需要保证 Op 的 Input、Output 和 Attribute 不能被修改（文档除外）或删除，可以新增 Input、Output 和 Attribute，但是新增的 Input，Output 必须设置 AsDispensable，新增的 Attribute 必须设置默认值。更多详细内容请参考OP 修改规范：Input/Output/Attribute 只能做兼容修改** 。

6.ShareDataWith 的调用

ShareDataWith 的功能是使两个 Tensor 共享底层 buffer，在调用这个操作的时候需要特别注意，在 Op 内部不能将 ShareDataWith 作用在 Op 的输出上，即 Op 输出的 Tensor 必须是 Malloc 出来的。

7. 稀疏梯度参数更新方法

目前稀疏梯度在做更新的时候会先对梯度做 merge，即对相同参数的梯度做累加，然后做参数以及附加参数（如 velocity）的更新。

8. 显存优化

8.1 为可原位计算的 Op 注册 Inplace

有些 Op 的计算逻辑中，输出可以复用输入的显存空间，也可称为原位计算。例如reshape_op中，输出 `Out` 可以复用输入 `X` 的显存空间，因为该 Op 的计算逻辑不会改变 `X` 的实际数据，只是修改它的 shape，输出和输入复用同一块显存空间不影响结果。对于这类 OP，可以注册 `Inlace`，从而让框架在运行时自动地进行显存优化。

Paddle 提供了 `DECLARE_INPLACE_OP_INFERER` 宏用于注册 `Inplace`，该宏第一个参数是一个类名，如 `ReshapeOpInplaceInToOut`；第二个参数是一对复用的输入输出，以 `{"X", "Out"}` 的形式给出。在 `REGISTER_OPERATOR` 时，可以将类名传传入，从而为该 Op 注册 `Inplace`。

```
DECLARE_INPLACE_OP_INFERER(ReshapeOpInplaceInToOut, {"X", "Out"});

REGISTER_OPERATOR(
    reshape, ops::ReshapeOp, ops::ReshapeOpMaker,
    paddle::framework::DefaultGradOpMaker<paddle::framework::OpDesc, true>,
    paddle::framework::DefaultGradOpMaker<paddle::imperative::OpBase, true>,
    ops::ReshapeOpInplaceInToOut);
```

8.2 减少 OP 中的无关变量

通常反向 Op 会依赖于前向 Op 的某些输入 (Input)、输出 (Output)，以供反向 Op 计算使用。但有些情况下，反向 Op 不需要前向 Op 的所有输入和输出；有些情况下，反向 Op 只需要前向 Op 的部分输入和输出；有些情况下，反向 Op 只需要使用前向 Op 中输入和输出变量的 Shape 和 LoD 信息。若 Op 开发者在注册反向 Op 时，将不必要的前向 Op 输入和输出作为反向 Op 的输入，会导致这部分显存无法被框架现有的显存优化策略优化，从而导致模型显存占用过高。

所以在写注册反向 Op 时需要注意以下几点：

- Paddle 提供的 `DefaultGradOpMaker`，默认会将前向 op 的所有输入 (`Input`) 、输出 (`Output`) 以及输出变量所对应的梯度 (`Output@Grad`) 作为反向 Op 的输入，将前向 Op 输入所对应的梯度 (`Input@Grad`) 作为反向 Op 的输出。所以在使用 `DefaultGradOpMaker` 时需要考虑是否有些变量在计算中不被用到。
- 如果 `DefaultGradOpMaker` 不能够满足需求，需要用户自己手动构建 `GradOpMaker`，具体实现请参考相关文档;
- 如果有些反向 Op 需要依赖前向 Op 的输入或输出变量的的 Shape 或 LoD，但不依赖于变量中 Tensor 的 Buffer，且不能根据其他变量推断出该 Shape 和 LoD，则可以通过 `DECLARE_NO_NEED_BUFFER_VARS_INFERER` 接口对该变量（以下称该变量为 `X`）在反向 Op 中进行注册 `NoNeedBufferVars`。**一旦注册了 `NoNeedBufferVars`，反向 op 中就不能读写该变量对应的 Tensor 中的 buffer，只能调用 Tensor 的 dims() 和 lod() 方法，同时，反向 Op 中的 `GetExpectedKernelType()` 必须要重写，并且 `GetExpectedKernelType()` 中不能访问 `X` 变量中 Tensor 的 type() 方法**。比如在 `SliceOpGrad` 中只会用到 `Input` 中变量的 Shape 信息，所以需要为对 `Input` 在 `SliceOpGrad` 上进行注册：

```
namespace paddle {
namespace operators {
// ...
class SliceOpGrad : public framework::OperatorWithKernel {
 public:
  using framework::OperatorWithKernel::OperatorWithKernel;

  void InferShape(framework::InferShapeContext* ctx) const override {
    // ...
  }

  framework::OpKernelType GetExpectedKernelType(
      const framework::ExecutionContext& ctx) const override {
    // Note: don't get data type from ctx.Input<framework::Tensor>("Input");
    auto dtype = ctx.Input<framework::Tensor>(framework::GradVarName("Out"))->type();
    return framework::OpKernelType( dtype, ctx.GetPlace());
  }
};
```

(下页继续)

(续上页)

```
template <typename T>
class SliceOpGradMaker : public framework::SingleGradOpMaker<T> {
 public:
  using framework::SingleGradOpMaker<T>::SingleGradOpMaker;

 protected:
  void Apply(GradOpPtr<T> bind) const override {
    bind->SetInput("Input", this->Input("Input"));
    if (this->HasInput("StartsTensor")) {
      bind->SetInput("StartsTensor", this->Input("StartsTensor"));
    }
    if (this->HasInput("EndsTensor")) {
      bind->SetInput("EndsTensor", this->Input("EndsTensor"));
    }
    if (this->HasInput("StartsTensorList")) {
      bind->SetInput("StartsTensorList", this->Input("StartsTensorList"));
    }
    if (this->HasInput("EndsTensorList")) {
      bind->SetInput("EndsTensorList", this->Input("EndsTensorList"));
    }
    bind->SetInput(framework::GradVarName("Out"), this->OutputGrad("Out"));
    bind->SetOutput(framework::GradVarName("Input"), this->InputGrad("Input"));
    bind->SetAttrMap(this->Attrs());
    bind->SetType("slice_grad");
  }
};

DECLARE_NO_NEED_BUFFER_VARS_INFERER(SliceOpGradNoNeedBufferVarsInference,
                                    "Input");
}  // namespace operators
}  // namespace paddle
namespace ops = paddle::operators;
REGISTER_OPERATOR(slice, ops::SliceOp, ops::SliceOpMaker,
                  ops::SliceOpGradMaker<paddle::framework::OpDesc>,
                  ops::SliceOpGradMaker<paddle::imperative::OpBase>);
REGISTER_OPERATOR(slice_grad, ops::SliceOpGrad,
                  ops::SliceDoubleOpGradMaker<paddle::framework::OpDesc>,
                  ops::SliceDoubleOpGradMaker<paddle::imperative::OpBase>,
                  ops::SliceOpGradNoNeedBufferVarsInference);
```

9. 混合设备调用

由于 GPU 是异步执行的，当 CPU 调用返回之后，GPU 端可能还没有真正的执行，所以如果在 Op 中创建了 GPU 运行时需要用到的临时变量，当 GPU 开始运行的时候，该临时变量可能在 CPU 端已经被释放，这样可能会导致 GPU 计算出错。

关于 GPU 中的一些同步和异步操作：

```
The following device operations are asynchronous with respect to the host:
    Kernel launches;
    Memory copies within a single device's memory;
    Memory copies from host to device of a memory block of 64 KB or less;
    Memory copies performed by functions that are suffixed with Async;
    Memory set function calls.
```

关于 cudaMemCpy 和 cudaMemCpyAsync 注意事项：

- 如果数据传输是从 GPU 端到非页锁定的 CPU 端，数据传输将是同步，即使调用的是异步拷贝操作。
- 如果数据传输是从 CPU 端到 CPU 端，数据传输将是同步的，即使调用的是异步拷贝操作。

更多内容可参考：Asynchronous Concurrent Execution，API synchronization behavior

10. LoD 在 Op 内部的传导规范

LoD 是 Paddle 框架用来表示变长序列数据的属性，除了仅支持输入是 padding data 的 Op 外，所有 Op 的实现都要考虑 LoD 的传导问题。

根据 OP 的计算过程中是否用到 LoD，我们可以将涉及到 LoD 传导问题的 OP 分为两类: LoD-Transparent 与 LoD-Based。

这两类 OP 的 LoD 传导需要考虑前向和反向两个过程。

前向传导

在前向传导过程，与输入的 LoD 相比较，Op 输出的 LoD 可能出现不变、改变和消失这三种情况：

- 不变：适用于所有的 LoD-Transparent OP 与部分的 LoD-Based OP。可以在 `InferShape` 中调用 `ShareLoD()` 直接将输入 Var 的 LoD 共享给输出 Var, 可参考 lstm_op; 如果有多个输入且都可能存在 LoD 的情况，通常默认共享第一个输入, 例如 elementwise_ops forward；
- 改变：适用于部分 LoD-Based OP。在实现 OpKernel 时需考虑输出 LoD 的正确计算，真实的 LoD 在前向计算结束后才能确定，此时仍需要在 `InferShape` 中调用 `ShareLoD()`，以确保 CompileTime 时对 LoD Level 做了正确的传导，可参考 sequence_expand_op；
- 消失：适用于输出不再是序列数据的 LoD-Based OP。此时不用再考虑前向的 LoD 传导问题，可参考 sequence_pool_op；

其它重要的注意事项：

- 实现 LoD-Based OP 时，需要处理好 LoD 传导的边界情况，例如对长度为零的输入的支持，并完善相应的单测，单测 case 覆盖空序列出现在 batch 开头、中间和末尾等位置的情况，可参考 test_lstm_op.py
- 对 LoD Level 有明确要求的 OP，推荐的做法是在 `InferShape` 中即完成 LoD Level 的检查，例如 sequence_pad_op。

反向传导

通常来讲，OP 的某个输入 Var 所对应的梯度 GradVar 的 LoD 应该与 Var 自身相同，所以应直接将 Var 的 LoD 共享给 GradVar，可以参考 elementwise ops 的 backward

8.2.3 Op 性能优化

1. 第三方库的选择

在写 Op 过程中优先使用高性能（如 cudnn、mkldnn、mklml、eigen 等）中提供的操作，但是一定要做 benchmark，有些库中的操作在深度学习任务中可能会比较慢。因为高性能库（如 eigen 等）中提供的操作为了更为通用，在性能方面可能并不是很好，通常深度学习模型中数据量较小，所以有些情况下可能高性能库中提供的某些操作速度较慢。比如 Elementwise 系列的所有 Op（前向和反向），Elementwise 操作在模型中调用的次数比较多，尤其是 Elementwise_add，在很多操作之后都需要添加偏置项。在之前的实现中 Elementwise_op 直接调用 Eigen 库，由于 Elementwise 操作在很多情况下需要对数据做 Broadcast，而实验发现 Eigen 库做 Broadcast 的速度比较慢，慢的原因在这个 PR#6229中有描述。

2.Op 性能优化

Op 的计算速度与输入的数据量有关，对于某些 Op 可以根据输入数据的 Shape 和 Op 的属性参数来选择不同的计算方式。比如 concat_op，当 axis>=1 时，在对多个 tensor 做拼接过程中需要对每个 tensor 做很多次拷贝，如果是在 GPU 上，需要调用 cudaMemCopy。相对 CPU 而言，GPU 属于外部设备，所以每次调用 GPU 的操作都会有一定的额外开销，并且当需要拷贝的次数较多时，这种开销就更为凸现。目前 concat_op 的实现会根据输入数据的 Shape 以及 axis 值来选择不同的调用方式，如果输入的 tensor 较多，且 axis 不等于 0，则将多次拷贝操作转换成一个 CUDA Kernel 来完成；如果输入 tensor 较少，且 axis 等于 0，使用直接进行拷贝。相关实验过程在该 PR（#8669）中有介绍。

由于 CUDA Kernel 的调用有一定的额外开销，所以如果 Op 中出现多次调用 CUDA Kernel，可能会影响 Op 的执行速度。比如之前的 sequence_expand_op 中包含很多 CUDA Kernel，通常这些 CUDA Kernel 处理的数据量较小，所以频繁调用这样的 Kernel 会影响 Op 的计算速度，这种情况下最好将这些小的 CUDA Kernel 合并成一个。在优化 sequence_expand_op 过程（相关 PR#9289）中就是采用这种思路，优化后的 sequence_expand_op 比之前的实现平均快出约 1 倍左右，相关实验细节在该 PR（#9289）中有介绍。

减少 CPU 与 GPU 之间的拷贝和同步操作的次数。比如 fetch 操作，在每个迭代之后都会对模型参数进行更新并得到一个 loss，并且数据从 GPU 端到没有页锁定的 CPU 端的拷贝是同步的，所以频繁的 fetch 多个参数

会导致模型训练速度变慢。

8.2.4 Op 数值稳定性问题

1. 有些 Op 存在数值稳定性问题

出现数值稳定性的主要原因程序在多次运行时，对浮点型数据施加操作的顺序可能不同，进而导致最终计算结果不同。而 GPU 是通过多线程并行计算的方式来加速计算的，所以很容易出现对浮点数施加操作的顺序不固定现象。

目前发现 cudnn 中的卷积操作、cudnn 中的 MaxPooling、CUDA 中 CudaAtomicXX、ParallelExecutor 的 Reduce 模式下参数梯度的聚合等操作运行结果是非确定的。

为此 Paddle 中添加了一些 FLAGS，比如使用 FLAGS_cudnn_deterministic 来强制 cudnn 使用确定性算法、FLAGS_cpu_deterministic 强制 CPU 端的计算使用确定性方法。

2.WITH_FAST_MATH 的开与关

如果 WITH_FAST_MATH 是 ON，NVCC 在编译 Paddle 和 Egien 的时候会使用–use_fast_math，这样可能会使 CUDA 中的一些操作在损失一定精度的情况下变快，比如 log、exp、tanh 等，但也会使一些操作的计算结果是错的，比如 pow 操作，具体原因请查看torch/DEPRECEATED-torch7-distro#132。

8.2.5 其他

1. 报错信息

Enforce 提示信息不能为空，并且需要写明，因为报错信息可以更快更方便地分析出错误的原因。

2.Op 的数学公式

如果 Op 有数学公式，一定要在代码中将数学公式写明，并在 Python API 的 Doc 中显示，因为用户在对比不同框架的计算结果时可能需要了解 Paddle 对 Op 是怎么实现的。

** 注意：** 在 merge 到 develop 分支之前一定进行公式预览。可参考dynamic_lstmp。

3.Op 变量名的命名要规范

在定义 Op 时，Op 的输入输出以及属性的命名需要符合规范，具体命名规则请参考：name_convention。

4.Python 端 Op 接口中参数的顺序

Python API 中参数的顺序一般按照重要性来排，以 fc 为例：

```
def fc(input,
       size,
       num_flatten_dims=1,
       param_attr=None,
       bias_attr=None,
       act=None,
       is_test=False,
       name=None)
```

8.3 如何写新的 Python OP

Paddle 通过 py_func 接口支持在 Python 端自定义 OP。py_func 的设计原理在于 Paddle 中的 Tensor 可以与 numpy 数组可以方便的互相转换，从而可以使用 Python 中的 numpy API 来自定义一个 Python OP。

8.3.1 py_func 接口概述

py_func 具体接口为：

```
def py_func(func, x, out, backward_func=None, skip_vars_in_backward_input=None):
    pass
```

其中，

- x 是 Python Op 的输入变量，可以是单个 Tensor|tuple[Tensor]|list[Tensor] 。多个 Tensor 以 tuple[Tensor] 或 list[Tensor] 的形式传入。
- out 是 Python Op 的输出变量，可以是单个 Tensor|tuple[Tensor]|list[Tensor]，也可以是 Numpy Array。
- func 是 Python Op 的前向函数。在运行网络前向时，框架会调用 out = func(*x) ，根据前向输入 x 和前向函数 func 计算前向输出 out。在 func 建议先主动将 Tensor 转换为 numpy 数组，方便灵活的使用 numpy 相关的操作，如果未转换成 numpy，则可能某些操作无法兼容。
- backward_func 是 Python Op 的反向函数。若 backward_func 为 None ，则该 Python Op 没有反向计算逻辑；若 backward_func 不为 None，则框架会在运行网路反向时调用 backward_func 计算前向输入 x 的梯度。
- skip_vars_in_backward_input 为反向函数 backward_func 中不需要的输入，可以是单个 Tensor|tuple[Tensor]|list[Tensor] 。

8.3.2 如何使用 py_func 编写 Python Op

以下以 tanh 为例，介绍如何利用 py_func 编写 Python Op。

- 第一步：定义前向函数和反向函数

前向函数和反向函数均由 Python 编写，可以方便地使用 Python 与 numpy 中的相关 API 来实现一个自定义的 OP。

若前向函数的输入为 x_1, x_2, ···, x_n ，输出为 y_1, y_2, ···, y_m，则前向函数的定义格式为：

```
def foward_func(x_1, x_2, ..., x_n):
    ...
    return y_1, y_2, ..., y_m
```

默认情况下，反向函数的输入参数顺序为：所有前向输入变量 + 所有前向输出变量 + 所有前向输出变量的梯度，因此对应的反向函数的定义格式为：

```
def backward_func(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m, dy_1, dy_2, ..., dy_m):
    ...
    return dx_1, dx_2, ..., dx_n
```

若反向函数不需要某些前向输入变量或前向输出变量，可设置 skip_vars_in_backward_input 进行排除（步骤三中会叙述具体的排除方法）。

注：， x_1, ···, x_n 为输入的多个 Tensor，请以 tuple(Tensor) 或 list[Tensor] 的形式在 py_func 中传入。建议先主动将 Tensor 通过 numpy.array 转换为数组，否则 Python 与 numpy 中的某些操作可能无法兼容使用在 Tensor 上。

此处我们利用 numpy 的相关 API 完成 tanh 的前向函数和反向函数编写。下面给出多个前向与反向函数定义的示例：

```
import numpy as np

# 前向函数 1：模拟 tanh 激活函数
def tanh(x):
    # 可以直接将 Tensor 作为 np.tanh 的输入参数
    return np.tanh(x)

# 前向函数 2：将两个 2-D Tenosr 相加，输入多个 Tensor 以 list[Tensor] 或 tuple(Tensor) 形式
def element_wise_add(x, y):
    # 必须先手动将 Tensor 转换为 numpy 数组，否则无法支持 numpy 的 shape 操作
    x = np.array(x)
    y = np.array(y)

    if x.shape != y.shape:
        raise AssertionError("the shape of inputs must be the same!")
```

(下页继续)

(续上页)

```
    result = np.zeros(x.shape, dtype='int32')
    for i in range(len(x)):
        for j in range(len(x[0])):
            result[i][j] = x[i][j] + y[i][j]

    return result

# 前向函数 3：可用于调试正在运行的网络（打印值）
def debug_func(x):
    # 可以直接将 Tensor 作为 print 的输入参数
    print(x)

# 前向函数 1 对应的反向函数，默认的输入顺序为：x、out、out 的梯度
def tanh_grad(x, y, dy):
    # 必须先手动将 Tensor 转换为 numpy 数组，否则"+/-" 等操作无法使用
    return np.array(dy) * (1 - np.square(np.array(y)))
```

注意，前向函数和反向函数的输入均是 Tensor 类型，输出可以是 Numpy Array 或 Tensor。由于 Tensor 实现了 Python 的 buffer protocol 协议，因此即可通过 numpy.array 直接将 Tensor 转换为 numpy Array 来进行操作，也可直接将 Tensor 作为 numpy 函数的输入参数。但建议先主动转换为 numpy Array，则可以任意的使用 python 与 numpy 中的所有操作（例如”numpy array 的 +/-/shape”）。

tanh 的反向函数不需要前向输入 x，因此我们可定义一个不需要前向输入 x 的反向函数，并在后续通过 skip_vars_in_backward_input 进行排除:

```
def tanh_grad_without_x(y, dy):
    return np.array(dy) * (1 - np.square(np.array(y)))
```

- 第二步：创建前向输出变量

我们需调用 Program.current_block().create_var 创建前向输出变量。在创建前向输出变量时，必须指明变量的名称 name、数据类型 dtype 和维度 shape。

```
import paddle

paddle.enable_static()

def create_tmp_var(program, name, dtype, shape):
    return program.current_block().create_var(name=name, dtype=dtype, shape=shape)

in_var = paddle.static.data(name='input', dtype='float32', shape=[-1, 28, 28])

# 手动创建前向输出变量
```

(下页继续)

(续上页)

```
out_var = create_tmp_var(paddle.static.default_main_program(), name='output', dtype=
↪'float32', shape=[-1, 28, 28])
```

- 第三步：调用 py_func 组建网络

py_func 的调用方式为:

```
paddle.static.nn.py_func(func=tanh, x=in_var, out=out_var, backward_func=tanh_grad)
```

若我们不希望在反向函数输入参数中出现前向输入，则可使用 skip_vars_in_backward_input 进行排查，简化反向函数的参数列表。

```
paddle.static.nn.py_func(func=tanh, x=in_var, out=out_var, backward_func=tanh_grad_
↪without_x,
    skip_vars_in_backward_input=in_var)
```

至此，使用 py_func 编写 Python Op 的步骤结束。我们可以与使用其他 Op 一样进行网路训练/预测。

8.3.3 注意事项

- py_func 的前向函数和反向函数内部不应调用 paddle.xx 组网接口，因为前向函数和反向函数是在网络运行时调用的，而 paddle.xx 是在组建网络的阶段调用。
- skip_vars_in_backward_input 只能跳过前向输入变量和前向输出变量，不能跳过前向输出的梯度。
- 若某个前向输出变量没有梯度，则 backward_func 将接收到 None 的输入。若某个前向输入变量没有梯度，则我们应在 backward_func 中主动返回 None。

8.4 如何在框架外部自定义 C++ OP

通常，如果 PaddlePaddle 的 Operator(OP) 库中没有您所需要的操作，建议先尝试使用已有的 OP 组合，如果无法组合出您需要的操作，可以尝试使用 paddle.static.py_func，也可以按照这篇教程自定义 C++ OP。当然，如果用若干 OP 组合出来的 OP 性能无法满足您的要求，也可以自定义 C++ OP。

自定义 OP 需要以下几个步骤:

1. 实现 OP 和注册 OP，和在框架内部写 OP 完全相同，遵守”如何写新的 C++ OP”的规范和步骤。当然，实现 Gradient OP 是可选的。
2. 编译出动态库。
3. 封装该 OP 的 Python 接口。
4. 写 OP 的单测。

下面通过一个具体的例子来详细的介绍，一步一步教会您如何实现。下面通过实现 relu op 来介绍。

8.4.1 自定义 OP 的实现

OP 的实现与”如何写新的 C++ OP”的教程相同，简答的说需要: 1). 定义 OP 的 ProtoMaker，即描述 OP 的输入、输出、属性信息；2). 实现 OP 的定义和 InferShape，以及 OP 的 kernel 函数，反向 OP 类似。3). 注册 OP，以及 OP 的计算函数。

ReLU OP 的 CPU 实现，`relu_op.cc` 文件:

```
// relu_op.cc
#include "paddle/fluid/framework/op_registry.h"

namespace paddle {
namespace operators {

// 前向 OP 的输入 X、输出 Y、属性
class Relu2OpMaker : public framework::OpProtoAndCheckerMaker {
 public:
  void Make() override {
    AddInput("X", "The input tensor.");
    AddOutput("Y", "Output of relu_op");
    AddComment(R"DOC(
Relu Operator.
Y = max(X, 0)
)DOC");
  }
};

// 前向 OP 的定义和 InferShape 实现，设置输出 Y 的 shape
class Relu2Op : public framework::OperatorWithKernel {
 public:
  using framework::OperatorWithKernel::OperatorWithKernel;

  void InferShape(framework::InferShapeContext* ctx) const override {
    auto in_dims = ctx->GetInputDim("X");
    ctx->SetOutputDim("Y", in_dims);
  }
};

// 实现前向 OP 的 Kernel 计算函数: Y = max(0, X)
using Tensor = framework::Tensor;
template <typename DeviceContext, typename T>
class Relu2Kernel : public framework::OpKernel<T> {
```

(下页继续)

(续上页)

```
 public:
  void Compute(const framework::ExecutionContext& ctx) const override {
    auto* in_t = ctx.Input<Tensor>("X");
    auto* out_t = ctx.Output<Tensor>("Y");
    auto x = in_t->data<T>();
    // mutable_data 分配内存、获取指针
    auto y = out_t->mutable_data<T>(ctx.GetPlace());
    for (int i = 0; i < in_t->numel(); ++i) {
      y[i] = std::max(static_cast<T>(0.), x[i]);
    }
  }
};

// 定义反向 OP 的输入 Y 和 dY、输出 dX、属性:
template <typename T>
class Relu2GradMaker : public framework::SingleGradOpMaker<T> {
 public:
  using framework::SingleGradOpMaker<T>::SingleGradOpMaker;

  void Apply(GradOpPtr<T> op) const override {
    op->SetType("relu2_grad");
    op->SetInput("Y", this->Output("Y"));
    op->SetInput(framework::GradVarName("Y"), this->OutputGrad("Y"));
    op->SetAttrMap(this->Attrs());
    op->SetOutput(framework::GradVarName("X"), this->InputGrad("X"));
  }
};

// 定义反向 OP 和 InferShape 实现, 设置 dX 的 shape
class Relu2GradOp : public framework::OperatorWithKernel {
 public:
  using framework::OperatorWithKernel::OperatorWithKernel;

  void InferShape(framework::InferShapeContext* ctx) const override {
    auto in_dims = ctx->GetInputDim(framework::GradVarName("Y"));
    ctx->SetOutputDim(framework::GradVarName("X"), in_dims);
  }
};

// 实现反向 OP 的 kernel 函数 dx = dy * ( y > 0. ? 1. : 0)
template <typename DeviceContext, typename T>
class Relu2GradKernel : public framework::OpKernel<T> {
 public:
```

(下页继续)

(续上页)

```
  void Compute(const framework::ExecutionContext& ctx) const override {
    auto* dy_t = ctx.Input<Tensor>(framework::GradVarName("Y"));
    auto* y_t = ctx.Input<Tensor>("Y");
    auto* dx_t = ctx.Output<Tensor>(framework::GradVarName("X"));

    auto dy = dy_t->data<T>();
    auto y = y_t->data<T>();
    auto dx = dx_t->mutable_data<T>(ctx.GetPlace());

    for (int i = 0; i < y_t->numel(); ++i) {
      dx[i] = dy[i] * (y[i] > static_cast<T>(0) ? 1. : 0.);
    }
  }
};

}  // namespace operators
}  // namespace paddle

namespace ops = paddle::operators;
using CPU = paddle::platform::CPUDeviceContext;
// 注册前向和反向 op
// 为了和框架内部的 relu 区分，这里注册的 OP type 为 relu2
REGISTER_OPERATOR(relu2,
                  ops::Relu2Op,
                  ops::Relu2OpMaker,
                  ops::Relu2GradMaker<paddle::framework::OpDesc>,
                  ops::Relu2GradMaker<paddle::imperative::OpBase>);
REGISTER_OPERATOR(relu2_grad, ops::Relu2GradOp);
// 注册 CPU 的 Kernel
REGISTER_OP_CPU_KERNEL(relu2,
                       ops::Relu2Kernel<CPU, float>,
                       ops::Relu2Kernel<CPU, double>);
REGISTER_OP_CPU_KERNEL(relu2_grad,
                       ops::Relu2GradKernel<CPU, float>,
                       ops::Relu2GradKernel<CPU, double>);
```

ReLU OP 的 GPU 实现，`relu_op.cu` 文件:

```
// relu_op.cu
#include "paddle/fluid/framework/op_registry.h"

namespace paddle {
namespace operators {
```

(下页继续)

(续上页)

```
using Tensor = framework::Tensor;

template <typename T>
__global__ void KeRelu2(const T* x, const int num, T* y) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = gid; i < num; i += blockDim.x * gridDim.x) {
    y[i] = max(x[i], static_cast<T>(0.));
  }
}

// 前向 OP 的 kernel 的 GPU 实现
template <typename DeviceContext, typename T>
class Relu2CUDAKernel : public framework::OpKernel<T> {
 public:
  void Compute(const framework::ExecutionContext& ctx) const override {
    auto* in_t = ctx.Input<Tensor>("X");
    auto* out_t = ctx.Output<Tensor>("Y");
    auto x = in_t->data<T>();
    auto y = out_t->mutable_data<T>(ctx.GetPlace());

    auto& dev_ctx = ctx.template device_context<DeviceContext>();

    int num = in_t->numel();
    int block = 512;
    int grid = (num + block - 1) / block;
    KeRelu2<T><<<grid, block, 0, dev_ctx.stream()>>>(x, num, y);
  }
};

template <typename T>
__global__ void KeRelu2Grad(const T* y, const T* dy, const int num, T* dx) {
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = gid; i < num; i += blockDim.x * gridDim.x) {
    dx[i] = dy[i] * (y[i] > 0 ? 1. : 0.);
  }
}

// 反向 OP 的 kernel 的 GPU 实现
template <typename DeviceContext, typename T>
class Relu2GradCUDAKernel : public framework::OpKernel<T> {
 public:
  void Compute(const framework::ExecutionContext& ctx) const override {
    auto* dy_t = ctx.Input<Tensor>(framework::GradVarName("Y"));
```

(下页继续)

(续上页)

```
    auto* y_t = ctx.Input<Tensor>("Y");
    auto* dx_t = ctx.Output<Tensor>(framework::GradVarName("X"));

    auto dy = dy_t->data<T>();
    auto y = y_t->data<T>();
    auto dx = dx_t->mutable_data<T>(ctx.GetPlace());

    auto& dev_ctx = ctx.template device_context<DeviceContext>();

    int num = dy_t->numel();
    int block = 512;
    int grid = (num + block - 1) / block;
    KeRelu2Grad<T><<<grid, block, 0, dev_ctx.stream()>>>(y, dy, num, dx);
  }
};

}  // namespace operators
}  // namespace paddle

using CUDA = paddle::platform::CUDADeviceContext;
// 注册前向的 GPU Kernel
REGISTER_OP_CUDA_KERNEL(relu2,
                        paddle::operators::Relu2CUDAKernel<CUDA, float>,
                        paddle::operators::Relu2CUDAKernel<CUDA, double>);
// 注册反向的 GPU Kernel
REGISTER_OP_CUDA_KERNEL(relu2_grad,
                        paddle::operators::Relu2GradCUDAKernel<CUDA, float>,
                        paddle::operators::Relu2GradCUDAKernel<CUDA, double>);
```

注意点:

1. OP 的 type 不能和 PaddlePaddle 已有的 OP type 相同，否则在 Python 中使用时会报错。

8.4.2 自定义 OP 的编译

需要将实现的 C++、CUDA 代码编译成动态库，下面通过 g++/nvcc 编译，当然您也可以写 Makefile 或者 CMake。

编译需要 include PaddlePaddle 的相关头文件，如上面代码 `paddle/fluid/framework/op_registry.h`，需要链接 PaddlePaddle 的 lib 库。可通过下面命令获取到:

```
# python
>>> import paddle
>>> print(paddle.sysconfig.get_include())
/paddle/pyenv/local/lib/python2.7/site-packages/paddle/include
```

(下页继续)

(续上页)

```
>>> print(paddle.sysconfig.get_lib())
/paddle/pyenv/local/lib/python2.7/site-packages/paddle/libs
```

下面命令可编译出动态库:

```
include_dir=$( python -c 'import paddle; print(paddle.sysconfig.get_include())' )
lib_dir=$( python -c 'import paddle; print(paddle.sysconfig.get_lib())' )

echo $include_dir
echo $lib_dir

# PaddlePaddel >=1.6.1, 仅需要 include ${include_dir} 和 ${include_dir}/third_party
nvcc relu_op.cu -c -o relu_op.cu.o -ccbin cc -DPADDLE_WITH_CUDA -DEIGEN_USE_GPU -
↪DPADDLE_USE_DSO -DPADDLE_WITH_MKLDNN -Xcompiler -fPIC -std=c++11 -Xcompiler -fPIC -
↪w --expt-relaxed-constexpr -O3 -DNVCC \
    -I ${include_dir} \
    -I ${include_dir}/third_party \

g++ relu_op.cc relu_op.cu.o -o relu2_op.so -shared -fPIC -std=c++11 -O3 -DPADDLE_WITH_
↪MKLDNN \
  -I ${include_dir} \
  -I ${include_dir}/third_party \
  -L /usr/local/cuda/lib64 \
  -L ${lib_dir} -lpaddle_framework -lcudart
```

注意点:

1. 通过 NVCC 编译 CUDA 源文件时，需要加编译选项 `-DPADDLE_WITH_CUDA -DEIGEN_USE_GPU -DPADDLE_USE_DSO`，在框架源码中会使用这些宏定义进行条件编译。用户自定义的 C++ OP 实现编译时，选项的开启状态需要和核心框架编译行为一致。如 `EIGEN_USE_GPU` 是使用 Eigen 数学库的 GPU 实现时需要增加的编译选项。

2. 如果飞桨安装包中不包含 MKLDNN 库，则需要去掉编译选项`-DPADDLE_WITH_MKLDNN`。核心框架源码中 (比如 tensor.h) 有使用此宏定义进行条件编译，该选项是否打开同样需要和核心框架编译行为保持一致。默认的飞桨安装包中含有 MKLDNN 库。

3. 可多个 OP 编译到同一个动态库中。

4. 通过 pip 方式安装的 PaddlePaddle 由 GCC 4.8 编译得到，由于 GCC 4.8 和 GCC 5 以上 **C++11 ABI 不兼容**，您编写的自定义 OP，需要通过 GCC 4.8 编译。若是 GCC 5 及以上的环境上使用自定义 OP，推荐使用Docker 安装 PaddlePaddle，使得编 Paddle 和编译自定义 OP 的 GCC 版本相同。

8.4.3 封装 Python Layer 接口

需要使用 paddle.incubate.load_op_library 接口调用加载动态库，使得 PaddlePaddle 的主进程中可以使用用户自定义的 OP。

```
# custom_op.py
import paddle.incubate as incubate
# 调用 load_op_library 加载动态库
incubate.load_op_library('relu2_op.so')

from paddle.incubate import LayerHelper

def relu2(x, name=None):
    # relu2 的 type 和在 OP 中定义的 type 相同
    helper = LayerHelper("relu2", **locals())
    # 创建输出 Variable
    out = helper.create_variable_for_type_inference(dtype=x.dtype)
    helper.append_op(type="relu2", inputs={"X": x}, outputs={"Y": out})
    return out
```

注意点:

1. 一个动态库只需使用 paddle.incubate.load_op_library 在 paddle import 之后加载一次即可。
2. Python 接口的封装和 PaddlePaddle 框架内部的封装相同，更多的示例也可以阅读源码中 python/paddle/fluid/layers/nn.py 的代码示例。

8.4.4 单测测试

可以写个简单的 Python 程序测试计算的正确性:

静态图模式

```
import numpy as np
import paddle
from custom_op import relu2

paddle.enable_static()
data = paddle.static.data(name='data', shape=[None, 32], dtype='float32')
relu = relu2(data)
use_gpu = True  # or False
paddle.set_device('gpu' if use_gpu else 'cpu')
exe = paddle.static.Executor()
```

(下页继续)

(续上页)

```
x = np.random.uniform(-1, 1, [4, 32]).astype('float32')
out, = exe.run(feed={'data': x}, fetch_list=[relu])
np.allclose(out, np.maximum(x, 0.))
```

动态图模式

```
import numpy as np
import paddle
from custom_op import relu2

use_gpu = True  # or False
paddle.set_device('gpu' if use_gpu else 'cpu')

x = np.random.uniform(-1, 1, [4, 32]).astype('float32')
t = paddle.to_tensor(x)
out = relu2(t)
np.allclose(out.numpy(), np.maximum(x, 0.))
```

接下来可以在模型中使用您自定义的 OP 了!

8.4.5 如何在 C++ 预测库中使用

暂时不支持在 C++ 预测库中使用，后续会补充在 C++ 预测库中的使用示例。

8.4.6 FAQ

1. Q: 如果出现类似错误: `relu2_op.so: cannot open shared object file: No such file or directory` 以及 `libpaddle_framework.so: cannot open shared object file: No such file or directory`。

   A: 需要将 `relu2_op.so` 所在路径以及 `libpaddle_framework.so` 路径 (即 `paddle.sysconfig.get_lib()` 得到路径) 设置到环境变量 LD_LIBRARY_PATH 中:

   ```
   # 假如 relu2_op.so 路径是: `paddle/test`, 对于 Linux 环境设置:
   export LD_LIBRARY_PATH=paddle/test:$( python -c 'import paddle; print(paddle.
   ↪sysconfig.get_lib())'):$LD_LIBRARY_PATH
   ```

CHAPTER 9

参与开发

9.1 本地开发指南

本文将指导您如何在本地进行代码开发

9.1.1 代码要求

- 代码注释请遵守 Doxygen 的样式。
- 确保编译器选项 `WITH_STYLE_CHECK` 已打开，并且编译能通过代码样式检查。
- 所有代码必须具有单元测试。
- 通过所有单元测试。
- 请遵守提交代码的一些约定。

以下教程将指导您提交代码。

9.1.2 Fork

跳转到PaddlePaddle GitHub 首页，然后单击 `Fork` 按钮，生成自己目录下的仓库，比如 https://github.com/USERNAME/Paddle。

9.1.3 克隆（Clone）

将远程仓库 clone 到本地：

```
 git clone https://github.com/USERNAME/Paddle
 cd Paddle
```

9.1.4 创建本地分支

Paddle 目前使用Git 流分支模型进行开发，测试，发行和维护，具体请参考 Paddle 分支规范。

所有的 feature 和 bug fix 的开发工作都应该在一个新的分支上完成，一般从 `develop` 分支上创建新分支。

使用 `git checkout -b` 创建并切换到新分支。

```
 git checkout -b my-cool-stuff
```

值得注意的是，在 checkout 之前，需要保持当前分支目录 clean，否则会把 untracked 的文件也带到新分支上，这可以通过 `git status` 查看。

9.1.5 使用 `pre-commit` 钩子

Paddle 开发人员使用 pre-commit 工具来管理 Git 预提交钩子。它可以帮助我们格式化源代码（C++，Python），在提交（commit）前自动检查一些基本事宜（如每个文件只有一个 EOL，Git 中不要添加大文件等）。

`pre-commit` 测试是 Travis-CI 中单元测试的一部分，不满足钩子的 PR 不能被提交到 Paddle，首先安装并在当前目录运行它：

```
 pip install pre-commit
 pre-commit install
```

Paddle 使用 `clang-format` 来调整 C/C++ 源代码格式，请确保 `clang-format` 版本在 3.8 以上。

注：通过 `pip install pre-commit` 和 `conda install -c conda-forge pre-commit` 安装的 `yapf` 稍有不同的，Paddle 开发人员使用的是 `pip install pre-commit`。

9.1.6 开始开发

在本例中，我删除了 README.md 中的一行，并创建了一个新文件。

通过 `git status` 查看当前状态，这会提示当前目录的一些变化，同时也可以通过 `git diff` 查看文件具体被修改的内容。

```
  git status
On branch test
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        test

no changes added to commit (use "git add" and/or "git commit -a")
```

9.1.7 编译和单元测试

关于编译 PaddlePaddle 的源码，请参见从源码编译 选择对应的操作系统。关于单元测试，可参考Op 单元测试 的运行方法。

9.1.8 提交（commit）

接下来我们取消对 README.md 文件的改变，然后提交新添加的 test 文件。

```
  git checkout -- README.md
  git status
On branch test
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        test

nothing added to commit but untracked files present (use "git add" to track)
  git add test
```

Git 每次提交代码，都需要写提交说明，这可以让其他人知道这次提交做了哪些改变，这可以通过 `git commit` 完成。

```
➜  git commit
CRLF end-lines remover...............................(no files to check)Skipped
yapf.................................................(no files to check)Skipped
Check for added large files..............................................Passed
Check for merge conflicts................................................Passed
Check for broken symlinks................................................Passed
Detect Private Key...................................(no files to check)Skipped
Fix End of Files.....................................(no files to check)Skipped
clang-formater.......................................(no files to check)Skipped
[my-cool-stuff c703c041] add test file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 233
```

9.1.9 保持本地仓库最新

在准备发起 Pull Request 之前，需要同步原仓库（https://github.com/PaddlePaddle/Paddle）最新的代码。

首先通过 `git remote` 查看当前远程仓库的名字。

```
➜  git remote
origin
➜  git remote -v
origin        https://github.com/USERNAME/Paddle (fetch)
origin        https://github.com/USERNAME/Paddle (push)
```

这里 origin 是我们 clone 的远程仓库的名字，也就是自己用户名下的 Paddle，接下来我们创建一个原始 Paddle 仓库的远程主机，命名为 upstream。

```
➜  git remote add upstream https://github.com/PaddlePaddle/Paddle
➜  git remote
origin
upstream
```

获取 upstream 的最新代码并更新当前分支。

```
➜  git fetch upstream
➜  git pull upstream develop
```

9.1.10 Push 到远程仓库

将本地的修改推送到 GitHub 上，也就是 https://github.com/USERNAME/Paddle。

```
# 推送到远程仓库 origin 的 my-cool-stuff 分支上
  git push origin my-cool-stuff
```

9.2 提交 PR 注意事项

9.2.1 建立 Issue 并完成 Pull Request

建立一个 Issue 描述问题，并记录它的编号。

切换到所建分支，然后点击 `New pull request`。

选择目标分支：

在 PR 的描述说明中，填写 `resolve #Issue 编号`可以在这个 PR 被 merge 后，自动关闭对应的 Issue，具体请见这里。

接下来等待 review，如果有需要修改的地方，参照上述步骤更新 origin 中的对应分支即可。

9.2.2 签署 CLA 协议和通过单元测试

签署 CLA

在首次向 PaddlePaddle 提交 Pull Request 时，您需要您签署一次 CLA(Contributor License Agreement) 协议，以保证您的代码可以被合入，具体签署方式如下：

- 请您查看 PR 中的 Check 部分，找到 license/cla，并点击右侧 detail，进入 CLA 网站
- 请您点击 CLA 网站中的“Sign in with GitHub to agree”，点击完成后将会跳转回您的 Pull Request 页面

通过单元测试

您在 Pull Request 中每提交一次新的 commit 后，会触发 CI 单元测试，请确认您的 commit message 中已加入必要的说明，请见提交（commit）

请您关注您 Pull Request 中的 CI 单元测试进程，它将会在几个小时内完成

您仅需要关注和自己提交的分支相关的 CI 项目，例如您向 develop 分支提交代码，则无需关注 release/1.1 一栏是否通过测试

当所需的测试后都出现了绿色的对勾，表示您本次 commit 通过了各项单元测试

如果所需的测试后出现了红色叉号，代表您本次的 commit 未通过某项单元测试，在这种情况下，请您点击 detail 查看报错详情，并将报错原因截图，以评论的方式添加在您的 Pull Request 中，我们的工作人员将帮您查看

9.2.3 删除远程分支

在 PR 被 merge 进主仓库后，我们可以在 PR 的页面删除远程仓库的分支。

也可以使用 `git push origin :分支名` 删除远程分支，如：

```
  git push origin :my-cool-stuff
```

9.2.4 删除本地分支

最后，删除本地分支。

```
# 切换到 develop 分支
  git checkout develop

# 删除 my-cool-stuff 分支
  git branch -D my-cool-stuff
```

至此，我们就完成了一次代码贡献的过程。

9.2.5 提交代码的一些约定

为了使评审人在评审代码时更好地专注于代码本身，请您每次提交代码时，遵守以下约定：

1）请保证 Travis-CI 中单元测试能顺利通过。如果没过，说明提交的代码存在问题，评审人一般不做评审。

2）提交 PUll Request 前：

- 请注意 commit 的数量：

原因：如果仅仅修改一个文件但提交了十几个 commit，每个 commit 只做了少量的修改，这会给评审人带来很大困扰。评审人需要逐一查看每个 commit 才能知道做了哪些修改，且不排除 commit 之间的修改存在相互覆盖的情况。

建议：每次提交时，保持尽量少的 commit，可以通过 `git commit --amend` 补充上次的 commit。对已经 Push 到远程仓库的多个 commit，可以参考squash commits after push。

- 请注意每个 commit 的名称：应能反映当前 commit 的内容，不能太随意。

3）如果解决了某个 Issue 的问题，请在该 PUll Request 的**第一个**评论框中加上：`fix #issue_number`，这样当该 PUll Request 被合并后，会自动关闭对应的 Issue。关键词包括：close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved，请选择合适的词汇。详细可参考Closing issues via commit messages。

此外，在回复评审人意见时，请您遵守以下约定：

1）评审人的每个意见都必须回复（这是开源社区的基本礼貌，别人帮了忙，应该说谢谢）：

- 对评审意见同意且按其修改完的，给个简单的 `Done` 即可；
- 对评审意见不同意的，请给出您自己的反驳理由。

2）如果评审意见比较多：

- 请给出总体的修改情况。
- 请采用start a review进行回复，而非直接回复的方式。原因是每个回复都会发送一封邮件，会造成邮件灾难。

9.3 FAQ

目录

- *FAQ*
  - *1. CLA 签署不成功，怎么办？*
  - *2. CI 没有触发，怎么办？*
  - *3. CI 随机挂，即错误信息与本 PR 无关，怎么办？*
  - *4. 如何修改 API.spec？*

9.3.1 1. CLA 签署不成功，怎么办？

由于 CLA 是第三方开源库，有时候会不稳定。如果确定自己已签署 CLA，但 CLA 没触发成功，可尝试：

- 关闭并重新开启本 PR，来重新触发 CLA。点击 `Close pull request` ，再点击 `Reopen pull request` ，并等待几分钟。
- 如果上述操作重复 2 次仍未生效，请重新提一个 PR 或评论区留言。

9.3.2 2. CI 没有触发，怎么办？

- 请在 commit 信息中添加正确的 CI 触发规则：
  - develop 分支请添加 `test=develop`
  - release 分支请添加如 `test=release/1.4` 来触发 release/1.4 分支
  - 文档预览请添加 `test=document_preview`

- 该 CI 触发规则以 commit 为单位，即对同一个 PR 来说，不管前面的 commit 是否已经添加，如果新 commit 想继续触发 CI，那么仍然需要添加。
- 添加 CI 触发规则后，仍有部分 CI 没有触发：请关闭并重新开启本 PR，来重新触发 CI。

9.3.3 3. CI 随机挂，即错误信息与本 PR 无关，怎么办？

由于 develop 分支代码的不稳定性，CI 可能会随机挂。如果确定 CI 错误和本 PR 无关，请在评论区贴上错误截图和错误链接。

9.3.4 4. 如何修改 API.spec？

为了保证 API 接口/文档的稳定性，我们对 API 进行了监控，即 API.spec 文件。修改方法请参考 diff_api.py 。

注意：提交 PR 后请查看下 diff，不要改到非本 PR 修改的 API 上。

CHAPTER 10

其他说明

您可以通过以下内容，了解更多飞桨框架的说明:

- 飞桨硬件支持 : 说明飞桨产品支持的硬件。
- 飞桨框架 API 映射表 : 说明飞桨框架 1.X 版本与飞桨框架 2.0 版本 API 对应关系。

10.1 硬件支持

飞桨各个产品支持的硬件信息如下:

10.1.1 PaddlePaddle

10.1.2 Paddle Inference

10.1.3 Paddle Lite

注意: 如果你想了解更多芯片支持的信息，请联系我们，邮箱为 Paddle-better@baidu.com。

10.2 飞桨框架 API 映射表

本文档基于 PaddlePaddle v1.X 梳理了常用 API 与 PaddlePaddle v2.0RC1 对应关系。可根据对应关系，快速熟悉 PaddlePaddle 2.0RC1 的接口使用。

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
0	paddle.fluid.BuildStrategy	paddle.static.BuildStrategy
1	paddle.fluid.CompiledProgram	paddle.static.CompiledProgram
2	paddle.fluid.cpu_places	paddle.static.cpu_places
3	paddle.fluid.CPUPlace	paddle.CPUPlace
4	paddle.fluid.cuda_places	paddle.static.cuda_places
5	paddle.fluid.CUDAPinnedPlace	paddle.CUDAPinnedPlace
6	paddle.fluid.CUDAPlace	paddle.CUDAPlace
7	paddle.fluid.default_main_program	paddle.static.default_main_program
8	paddle.fluid.default_startup_program	paddle.static.default_startup_program
9	paddle.fluid.disable_dygraph	paddle.enable_static
10	paddle.fluid.embedding	paddle.nn.functional.embedding(动态图), paddle.static.nn.embedding(静态图)
11	paddle.fluid.enable_dygraph	paddle.disable_static
12	paddle.fluid.enable_imperative	paddle.disable_static
13	paddle.fluid.ExecutionStrategy	paddle.static.ExecutionStrategy
14	paddle.fluid.Executor	paddle.static.Executor
15	paddle.fluid.global_scope	paddle.static.global_scope
16	paddle.fluid.gradients	paddle.static.gradients
17	paddle.fluid.in_dygraph_mode	paddle.in_dynamic_mode
18	paddle.fluid.is_compiled_with_cuda	paddle.is_compiled_with_cuda
19	paddle.fluid.load	paddle.static.load
20	paddle.fluid.load_op_library	paddle.utils.load_op_library
21	paddle.fluid.name_scope	paddle.static.name_scope
22	paddle.fluid.one_hot	paddle.nn.functional.one_hot
23	paddle.fluid.ParallelExecutor	paddle.static.ParallelExecutor
24	paddle.fluid.ParamAttr	paddle.ParamAttr
25	paddle.fluid.Program	paddle.static.Program
26	paddle.fluid.program_guard	paddle.static.program_guard
27	paddle.fluid.require_version	paddle.utils.require_version
28	paddle.fluid.save	paddle.save
29	paddle.fluid.scope_guard	paddle.static.scope_guard
30	paddle.fluid.Variable	paddle.static.Variable

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
31	paddle.fluid.WeightNormParamAttr	paddle.static.WeightNormParamAttr
32	paddle.fluid.backward.append_backward	paddle.static.append_backward
33	paddle.fluid.backward.gradients	paddle.static.gradients
34	paddle.fluid.clip.GradientClipByGlobalNorm	paddle.nn.ClipGradByGlobalNorm
35	paddle.fluid.clip.GradientClipByNorm	paddle.nn.ClipGradByNorm
36	paddle.fluid.clip.GradientClipByValue	paddle.nn.ClipGradByValue
37	paddle.fluid.dataset.InMemoryDataset	paddle.distributed.InMemoryDataset
38	paddle.fluid.dataset.QueueDataset	paddle.distributed.QueueDataset
39	paddle.fluid.dygraph.BatchNorm	paddle.nn.BatchNorm1D, paddle.nn.BatchNorm2D, paddle.nn.BatchNorm3D
40	paddle.fluid.dygraph.BCELoss	paddle.nn.BCELoss
41	paddle.fluid.dygraph.BilinearTensorProduct	paddle.nn.Bilinear
42	paddle.fluid.dygraph.Conv2D	paddle.nn.Conv2D
43	paddle.fluid.dygraph.Conv2DTranspose	paddle.nn.Conv2DTranspose
44	paddle.fluid.dygraph.Conv3D	paddle.nn.Conv3D
45	paddle.fluid.dygraph.Conv3DTranspose	paddle.nn.Conv3DTranspose
46	paddle.fluid.dygraph.CosineDecay	paddle.optimizer.lr.CosineAnnealingDecay
47	paddle.fluid.dygraph.DataParallel	paddle.DataParallel
48	paddle.fluid.dygraph.disable_dygraph	paddle.enable_static
49	paddle.fluid.dygraph.Dropout	paddle.nn.Dropout, paddle.nn.Dropout2D, paddle.nn.Dropout3D
50	paddle.fluid.dygraph.Embedding	paddle.nn.Embedding
51	paddle.fluid.dygraph.enable_dygraph	paddle.disable_static
52	paddle.fluid.dygraph.enable_imperative	paddle.disable_static
53	paddle.fluid.dygraph.grad	paddle.grad
54	paddle.fluid.dygraph.GroupNorm	paddle.nn.GroupNorm
55	paddle.fluid.dygraph.InstanceNorm	paddle.nn.InstanceNorm1D, paddle.nn.InstanceNorm2D, paddle.nn.InstanceNorm3D
56	paddle.fluid.dygraph.L1Loss	paddle.nn.L1Loss
57	paddle.fluid.dygraph.Layer	paddle.nn.Layer
58	paddle.fluid.dygraph.LayerList	paddle.nn.LayerList
59	paddle.fluid.dygraph.LayerNorm	paddle.nn.LayerNorm
60	paddle.fluid.dygraph.Linear	paddle.nn.Linear
61	paddle.fluid.dygraph.load_dygraph	paddle.load
62	paddle.fluid.dygraph.MSELoss	paddle.nn.MSELoss
63	paddle.fluid.dygraph.NaturalExpDecay	paddle.optimizer.lr.NaturalExpDecay
64	paddle.fluid.dygraph.NLLLoss	paddle.nn.NLLLoss

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
65	paddle.fluid.dygraph.NoamDecay	paddle.optimizer.lr.NoamDecay
66	paddle.fluid.dygraph.ParameterList	paddle.nn.ParameterList
67	paddle.fluid.dygraph.no_grad	paddle.no_grad
68	paddle.fluid.dygraph.PolynomialDecay	paddle.optimizer.lr.PolynomialDecay
69	paddle.fluid.dygraph.Pool2D	paddle.nn.MaxPool2D, paddle.nn.AvgPool2D
70	paddle.fluid.dygraph.PRelu	paddle.nn.PReLU
71	paddle.fluid.dygraph.ProgramTranslator	paddle.jit.ProgramTranslator
72	paddle.fluid.dygraph.Sequential	paddle.nn.Sequential
73	paddle.fluid.dygraph.SpectralNorm	paddle.nn.SpectralNorm
74	paddle.fluid.dygraph.to_variable	paddle.to_tensor
75	paddle.fluid.dygraph.TracedLayer	paddle.jit.TracedLayer
76	paddle.fluid.executor.Executor	paddle.static.Executor
77	paddle.fluid.executor.global_scope	paddle.static.global_scope
78	paddle.fluid.executor.scope_guard	paddle.static.scope_guard
79	paddle.fluid.initializer.Bilinear	paddle.nn.initializer.Bilinear
80	paddle.fluid.initializer.BilinearInitializer	paddle.nn.initializer.Bilinear
81	paddle.fluid.initializer.Constant	paddle.nn.initializer.Constant
82	paddle.fluid.initializer.ConstantInitializer	paddle.nn.initializer.Constant
83	paddle.fluid.initializer.MSRA	paddle.nn.initializer.KaimingNormal, paddle.nn.initializer.KaimingUniform
84	paddle.fluid.initializer.MSRAInitializer	paddle.nn.initializer.KaimingNormal, paddle.nn.initializer.KaimingUniform
85	paddle.fluid.initializer.Normal	paddle.nn.initializer.Normal
86	paddle.fluid.initializer.NormalInitializer	paddle.nn.initializer.Normal
87	paddle.fluid.initializer.NumpyArrayInitializer	paddle.nn.initializer.Assign
88	paddle.fluid.initializer.TruncatedNormal	paddle.nn.initializer.TruncatedNormal
89	paddle.fluid.initializer.TruncatedNormalInitializer	paddle.nn.initializer.TruncatedNormal
90	paddle.fluid.initializer.Uniform	paddle.nn.initializer.Uniform
91	paddle.fluid.initializer.UniformInitializer	paddle.nn.initializer.Uniform
92	paddle.fluid.initializer.Xavier	paddle.nn.initializer.XavierNormal, paddle.nn.initializer.XavierUniform
93	paddle.fluid.initializer.XavierInitializer	paddle.nn.initializer.XavierNormal, paddle.nn.initializer.XavierUniform
94	paddle.fluid.io.DataLoader	paddle.io.DataLoader
95	paddle.fluid.io.load	paddle.static.load
96	paddle.fluid.io.load_inference_model	paddle.static.load_inference_model
97	paddle.fluid.io.load_program_state	paddle.static.load_program_state

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
98	paddle.fluid.io.save	paddle.save, paddle.fluid.io.save, paddle.static.save
99	paddle.fluid.io.save_inference_model	paddle.static.save_inference_model
100	paddle.fluid.io.set_program_state	paddle.static.set_program_state
101	paddle.fluid.layers.abs	paddle.abs
102	paddle.fluid.layers.accuracy	paddle.metric.accuracy
103	paddle.fluid.layers.acos	paddle.acos
104	paddle.fluid.layers.adaptive_pool2d	paddle.nn.functional.adaptive_avg_pool2d, paddle.nn.functional.adaptive_max_pool2d
105	paddle.fluid.layers.adaptive_pool3d	paddle.nn.functional.adaptive_max_pool3d, paddle.nn.functional.adaptive_avg_pool3d
106	paddle.fluid.layers.addcmul	paddle.tensor.math.addcmul
107	paddle.fluid.layers.addmm	paddle.addmm
108	paddle.fluid.layers.affine_grid	paddle.nn.functional.affine_grid
109	paddle.fluid.layers.allclose	paddle.allclose
110	paddle.fluid.layers.arange	paddle.arange
111	paddle.fluid.layers.argmax	paddle.argmax
112	paddle.fluid.layers.argmin	paddle.argmin
113	paddle.fluid.layers.argsort	paddle.argsort
114	paddle.fluid.layers.asin	paddle.asin
115	paddle.fluid.layers.atan	paddle.atan
116	paddle.fluid.layers.auc	paddle.metric.Auc
117	paddle.fluid.layers.batch_norm	paddle.static.nn.batch_norm
118	paddle.fluid.layers.bilinear_tensor_product	paddle.nn.functional.bilinear
119	paddle.fluid.layers.bmm	paddle.bmm
120	paddle.fluid.layers.case	paddle.static.nn.case
121	paddle.fluid.layers.cast	paddle.cast
122	paddle.fluid.layers.Categorical	paddle.distribution.Categorical
123	paddle.fluid.layers.ceil	paddle.ceil
124	paddle.fluid.layers.chunk_eval	paddle.metric.chunk_eval
125	paddle.fluid.layers.clamp	paddle.clip
126	paddle.fluid.layers.clip_by_norm	paddle.nn.clip_by_norm
127	paddle.fluid.layers.concat	paddle.concat
128	paddle.fluid.layers.cond	paddle.static.nn.cond
129	paddle.fluid.layers.conv2d	paddle.nn.functional.conv2d(动 态 图), paddle.static.nn.conv2d(静态图),
130	paddle.fluid.layers.conv2d_transpose	paddle.nn.functional.conv2d_transpose(动 态 图), paddle.static.nn.conv2d_transpose(静态图)

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
131	paddle.fluid.layers.conv3d	paddle.nn.functional.conv3d(动 态 图), paddle.static.nn.conv3d(静态图)
132	paddle.fluid.layers.conv3d_transpose	paddle.nn.functional.conv3d_transpose(动 态 图), paddle.static.nn.conv3d_transpose(静态图)
133	paddle.fluid.layers.cos	paddle.cos
134	paddle.fluid.layers.cos_sim	paddle.nn.functional.cosine_similarity
135	paddle.fluid.layers.create_parameter	paddle.create_parameter
136	paddle.fluid.layers.crf_decoding	paddle.static.nn.crf_decoding
137	paddle.fluid.layers.crop	paddle.crop
138	paddle.fluid.layers.cross	paddle.cross
139	paddle.fluid.layers.cumsum	paddle.cumsum
140	paddle.fluid.layers.data	paddle.static.data
141	paddle.fluid.layers.data_norm	paddle.static.nn.data_norm
142	paddle.fluid.layers.deformable_conv	paddle.static.nn.deform_conv2d
143	paddle.fluid.layers.diag	paddle.diag
144	paddle.fluid.layers.diag_embed	paddle.nn.functional.diag_embed
145	paddle.fluid.layers.dice_loss	paddle.nn.functional.dice_loss
146	paddle.fluid.layers.dist	paddle.dist
147	paddle.fluid.layers.dot	paddle.dot
148	paddle.fluid.layers.dropout	paddle.nn.functional.dropout, paddle.nn.functional.dropout2d, paddle.nn.functional.dropout3d
149	paddle.fluid.layers.dynamic_gru	paddle.nn.GRU
150	paddle.fluid.layers.dynamic_decode	paddle.nn.dynamic_decode
151	paddle.fluid.layers.elementwise_add	paddle.add
152	paddle.fluid.layers.elementwise_div	paddle.divide
153	paddle.fluid.layers.elementwise_equal	paddle.equal
154	paddle.fluid.layers.elementwise_floordiv	paddle.floor_divide
155	paddle.fluid.layers.elementwise_max	paddle.maximum
156	paddle.fluid.layers.elementwise_min	paddle.minimum
157	paddle.fluid.layers.elementwise_mod	paddle.mod
158	paddle.fluid.layers.elementwise_mul	paddle.multiply
159	paddle.fluid.layers.elu	paddle.nn.functional.elu
160	paddle.fluid.layers.embedding	paddle.nn.functional.embedding(动 态 图), paddle.static.nn.embedding(静态图)
161	paddle.fluid.layers.erf	paddle.erf
162	paddle.fluid.layers.exp	paddle.exp

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
163	paddle.fluid.layers.expand	paddle.expand
164	paddle.fluid.layers.expand_as	paddle.expand_as
165	paddle.fluid.layers.exponential_decay	paddle.optimizer.lr.ExponentialDecay
166	paddle.fluid.layers.eye	paddle.eye
167	paddle.fluid.layers.fc	paddle.nn.functional.linear(动 态 图), paddle.static.nn.fc(静态图)
168	paddle.fluid.layers.flatten	paddle.flatten
169	paddle.fluid.layers.flip	paddle.flip
170	paddle.fluid.layers.floor	paddle.floor
171	paddle.fluid.layers.full_like	paddle.full_like
172	paddle.fluid.layers.gather	paddle.gather
173	paddle.fluid.layers.gather_nd	paddle.gather_nd
174	paddle.fluid.layers.gelu	paddle.nn.functional.gelu
175	paddle.fluid.layers.greater_equal	paddle.greater_equal
176	paddle.fluid.layers.greater_than	paddle.greater_than
177	paddle.fluid.layers.group_norm	paddle.static.nn.group_norm
178	paddle.fluid.layers.GRUCell	paddle.nn.GRUCell
179	paddle.fluid.layers.hard_shrink	paddle.nn.functional.hardshrink
180	paddle.fluid.layers.hard_sigmoid	paddle.nn.functional.hardsigmoid
181	paddle.fluid.layers.hard_swish	paddle.nn.functional.hardswish
182	paddle.fluid.layers.has_inf	paddle.isinf
183	paddle.fluid.layers.has_nan	paddle.isnan
184	paddle.fluid.layers.hsigmoid	paddle.nn.functional.hsigmoid_loss
185	paddle.fluid.layers.increment	paddle.increment
186	paddle.fluid.layers.inverse_time_decay	paddle.optimizer.lr.InverseTimeDecay
187	paddle.fluid.layers.index_select	paddle.index_select
188	paddle.fluid.layers.instance_norm	paddle.static.nn.instance_norm
189	paddle.fluid.layers.interpolate	paddle.nn.functional.interpolate
190	paddle.fluid.layers.is_empty	paddle.is_empty
191	paddle.fluid.layers.isfinite	paddle.isfinite
192	paddle.fluid.layers.kldiv_loss	paddle.nn.functional.kl_div
193	paddle.fluid.layers.kron	paddle.kron
194	paddle.fluid.layers.label_smooth	paddle.nn.functional.label_smooth
195	paddle.fluid.layers.layer_norm	paddle.static.nn.layer_norm
196	paddle.fluid.layers.leaky_relu	paddle.nn.functional.leaky_relu
197	paddle.fluid.layers.less_equal	paddle.less_equal

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
198	paddle.fluid.layers.less_than	paddle.less_than
199	paddle.fluid.layers.linspace	paddle.linspace
200	paddle.fluid.layers.log	paddle.log
201	paddle.fluid.layers.log1p	paddle.log1p
202	paddle.fluid.layers.log_loss	paddle.nn.functional.log_loss
203	paddle.fluid.layers.log_softmax	paddle.nn.functional.log_softmax
204	paddle.fluid.layers.logical_and	paddle.logical_and
205	paddle.fluid.layers.logical_not	paddle.logical_not
206	paddle.fluid.layers.logical_or	paddle.logical_or
207	paddle.fluid.layers.logical_xor	paddle.logical_xor
208	paddle.fluid.layers.logsigmoid	paddle.nn.functional.log_sigmoid
209	paddle.fluid.layers.logsumexp	paddle.logsumexp
210	paddle.fluid.layers.lrn	paddle.nn.functional.local_response_norm
211	paddle.fluid.layers.lstm	paddle.nn.LSTM
212	paddle.fluid.layers.margin_rank_loss	paddle.nn.functional.margin_ranking_loss
213	paddle.fluid.layers.maxout	paddle.nn.functional.maxout
214	paddle.fluid.layers.mean_iou	paddle.metric.mean_iou
215	paddle.fluid.layers.meshgrid	paddle.meshgrid
216	paddle.fluid.layers.mse_loss	paddle.nn.functional.mse_loss
217	paddle.fluid.layers.mul	paddle.matmul
218	paddle.fluid.layers.multi_box_head	paddle.static.nn.multi_box_head
219	paddle.fluid.layers.multiplex	paddle.multiplex
220	paddle.fluid.layers.nce	paddle.static.nn.nce
221	paddle.fluid.layers.nonzero	paddle.nonzero
222	paddle.fluid.layers.Normal	paddle.distribution.Normal
223	paddle.fluid.layers.not_equal	paddle.not_equal
224	paddle.fluid.layers.npair_loss	paddle.nn.functional.npair_loss
225	paddle.fluid.layers.one_hot	paddle.nn.functional.one_hot
226	paddle.fluid.layers.ones	paddle.ones
227	paddle.fluid.layers.ones_like	paddle.ones_like
228	paddle.fluid.layers.pad2d	paddle.nn.functional.pad
229	paddle.fluid.layers.piecewise_decay	paddle.optimizer.lr.PiecewiseDecay
230	paddle.fluid.layers.pixel_shuffle	paddle.nn.functional.pixel_shuffle
231	paddle.fluid.layers.pool2d	paddle.nn.functional.avg_pool2d, paddle.nn.functional.max_pool2d
232	paddle.fluid.layers.pool3d	paddle.nn.functional.avg_pool3d, paddle.nn.functional.max_pool3d

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
233	paddle.fluid.layers.pow	paddle.pow
234	paddle.fluid.layers.prelu	paddle.nn.functional.prelu(动 态 图), paddle.static.nn.prelu(静态图)
235	paddle.fluid.layers.Print	paddle.static.Print
236	paddle.fluid.layers.py_func	paddle.static.py_func
237	paddle.fluid.layers.randint	paddle.randint
238	paddle.fluid.layers.randn	paddle.randn
239	paddle.fluid.layers.random_crop	paddle.vision.RandomCrop
240	paddle.fluid.layers.randperm	paddle.randperm
241	paddle.fluid.layers.rank	paddle.rank
242	paddle.fluid.layers.reciprocal	paddle.reciprocal
243	paddle.fluid.layers.reduce_all	paddle.all
244	paddle.fluid.layers.reduce_any	paddle.any
245	paddle.fluid.layers.reduce_max	paddle.max
246	paddle.fluid.layers.reduce_mean	paddle.mean
247	paddle.fluid.layers.reduce_min	paddle.min
248	paddle.fluid.layers.reduce_prod	paddle.prod
249	paddle.fluid.layers.reduce_sum	paddle.sum
250	paddle.fluid.layers.relu	paddle.nn.functional.relu
251	paddle.fluid.layers.relu6	paddle.nn.functional.relu6
252	paddle.fluid.layers.reshape	paddle.reshape
253	paddle.fluid.layers.rnn	paddle.nn.RNN
254	paddle.fluid.layers.roll	paddle.roll
255	paddle.fluid.layers.round	paddle.round
256	paddle.fluid.layers.rsqrt	paddle.rsqrt
257	paddle.fluid.layers.RNNCell	paddle.nn.RNNCellBase
258	paddle.fluid.layers.scale	paddle.scale
259	paddle.fluid.layers.scatter	paddle.scatter
260	paddle.fluid.layers.scatter_nd_add	paddle.scatter_nd_add
261	paddle.fluid.layers.scatter_nd	paddle.scatter_nd
262	paddle.fluid.layers.selu	paddle.nn.functional.selu
263	paddle.fluid.layers.shape	paddle.shape
264	paddle.fluid.layers.shard_index	paddle.shard_index
265	paddle.fluid.layers.sigmoid	paddle.nn.functional.sigmoid
266	paddle.fluid.layers.sigmoid_cross_entropy_with_logits	paddle.nn.functional.binary_cross_entropy
267	paddle.fluid.layers.sigmoid_focal_loss	paddle.nn.functional.sigmoid_focal_loss

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
268	paddle.fluid.layers.sign	paddle.sign
269	paddle.fluid.layers.sin	paddle.sin
270	paddle.fluid.layers.size	paddle.numel
271	paddle.fluid.layers.slice	paddle.slice
272	paddle.fluid.layers.smooth_l1	paddle.nn.functional.smooth_l1_loss
273	paddle.fluid.layers.softmax	paddle.nn.functional.softmax
274	paddle.fluid.layers.softmax_with_cross_entropy	paddle.nn.functional.cross_entropy
275	paddle.fluid.layers.softplus	paddle.nn.functional.softplus
276	paddle.fluid.layers.softshrink	paddle.nn.functional.softshrink
277	paddle.fluid.layers.softsign	paddle.nn.functional.softsign
278	paddle.fluid.layers.spectral_norm	paddle.static.nn.spectral_norm
279	paddle.fluid.layers.split	paddle.split
280	paddle.fluid.layers.sqrt	paddle.sqrt
281	paddle.fluid.layers.square	paddle.square
282	paddle.fluid.layers.square_error_cost	paddle.nn.functional.square_error_cost
283	paddle.fluid.layers.squeeze	paddle.squeeze
284	paddle.fluid.layers.stack	paddle.stack
285	paddle.fluid.layers.stanh	paddle.stanh
286	paddle.fluid.layers.strided_slice	paddle.strided_slice
287	paddle.fluid.layers.sums	paddle.add_n
288	paddle.fluid.layers.swish	paddle.nn.functional.swish
289	paddle.fluid.layers.switch_case	paddle.static.nn.switch_case
290	paddle.fluid.layers.t	paddle.t
291	paddle.fluid.layers.tanh	paddle.tanh
292	paddle.fluid.layers.tanh_shrink	paddle.nn.functional.tanhshrink
293	paddle.fluid.layers.thresholded_relu	paddle.nn.functional.thresholded_relu
294	paddle.fluid.layers.topk	paddle.topk
295	paddle.fluid.layers.trace	paddle.trace
296	paddle.fluid.layers.transpose	paddle.transpose
297	paddle.fluid.layers.tril	paddle.tril
298	paddle.fluid.layers.triu	paddle.triu
299	paddle.fluid.layers.unfold	paddle.nn.functional.unfold
300	paddle.fluid.layers.Uniform	paddle.distribution.Uniform
301	paddle.fluid.layers.unique	paddle.unique
302	paddle.fluid.layers.unsqueeze	paddle.unsqueeze
303	paddle.fluid.layers.unstack	paddle.unstack

下页继续

表 1 – 续上页

序号	PaddlePaddle 1.X API	PaddlePaddle 2.0RC1 API
304	paddle.fluid.layers.warpctc	paddle.nn.functional.ctc_loss
305	paddle.fluid.layers.where	paddle.where
306	paddle.fluid.layers.while_loop	paddle.static.nn.while_loop
307	paddle.fluid.layers.zeros	paddle.zeros
308	paddle.fluid.layers.zeros_like	paddle.zeros_like
309	paddle.fluid.metrics.Accuracy	paddle.metric.Accuracy
310	paddle.fluid.metrics.Precision	paddle.metric.Precision
311	paddle.fluid.metrics.Recall	paddle.metric.Recall
312	paddle.fluid.optimizer.Adadelta	paddle.optimizer.Adadelta
313	paddle.fluid.optimizer.AdadeltaOptimizer	paddle.optimizer.Adadelta
314	paddle.fluid.optimizer.Adagrad	paddle.optimizer.Adagrad
315	paddle.fluid.optimizer.AdagradOptimizer	paddle.optimizer.Adagrad
316	paddle.fluid.optimizer.Adam	paddle.optimizer.Adam
317	paddle.fluid.optimizer.Adamax	paddle.optimizer.Adamax
318	paddle.fluid.optimizer.AdamaxOptimizer	paddle.optimizer.Adamax
319	paddle.fluid.optimizer.AdamOptimizer	paddle.optimizer.Adam
320	paddle.fluid.optimizer.Momentum	paddle.optimizer.Momentum
321	paddle.fluid.optimizer.MomentumOptimizer	paddle.optimizer.Momentum
322	paddle.fluid.optimizer.RMSPropOptimizer	paddle.optimizer.RMSProp
323	paddle.fluid.optimizer.SGD	paddle.optimizer.SGD
324	paddle.fluid.optimizer.SGDOptimizer	paddle.optimizer.SGD
325	paddle.fluid.regularizer.L1Decay	paddle.regularizer.L1Decay
326	paddle.fluid.regularizer.L1DecayRegularizer	paddle.regularizer.L1Decay
327	paddle.fluid.regularizer.L2Decay	paddle.regularizer.L2Decay
328	paddle.fluid.regularizer.L2DecayRegularizer	paddle.regularizer.L2Decay