Fluid编程指南

本文档将指导您如何用Fluid API编程并搭建一个简单的神经网络。阅读完本文档，您将掌握：

- Fluid有哪些核心概念
- 如何在fluid中定义运算过程
- 如何使用executor运行fluid操作
- 如何从逻辑层对实际问题建模
- 如何调用API（层，数据集，损失函数，优化方法等等）

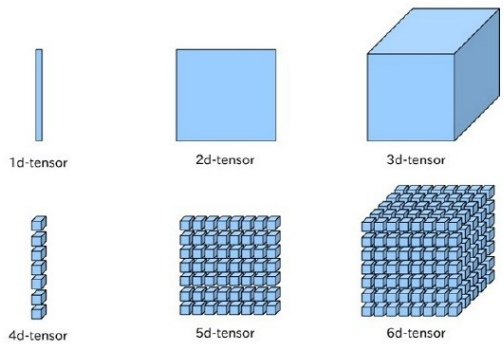
在进行模型搭建之前，首先需要明确几个Fluid核心使用概念：

使用Tensor表示数据

Fluid和其他主流框架一样，使用Tensor数据结构来承载数据。

在神经网络中传递的数据都是Tensor,Tensor可以简单理解成一个多维数组，一般而言可以有任意多的维度。不同的Tensor可以具有自己的数据类型和形状，同一Tensor中每个元素的数据类型是一样的，Tensor的形状就是Tensor的维度。

下图直观地表示1～6维的Tensor：



在 Fluid 中存在三种特殊的 Tensor：

1. **模型中的可学习参数**

模型中的可学习参数（包括网络权重、偏置等）生存期和整个训练任务一样长，会接受优化算法的更新，在 Fluid 中以 Variable 的子类 Parameter 表示。

在Fluid中可以通过 `fluid.layers.create_parameter` 来创建可学习参数：

```
w = fluid.layers.create_parameter(name="w",shape=[1],dtype='float32')
```

一般情况下，您不需要自己来创建网络中的可学习参数，Fluid 为大部分常见的神经网络基本计算模块都提供了封装。以最简单的全连接模型为例，下面的代码片段会直接为全连接层创建连接权值（W）和偏置（bias）两个可学习参数，无需显式地调用 Parameter 相关接口来创建。

```
import paddle.fluid as fluid
y = fluid.layers.fc(input=x, size=128, bias_attr=True)
```

2. **输入输出Tensor**

整个神经网络的输入数据也是一个特殊的 Tensor，在这个 Tensor 中，一些维度的大小在定义模型时无法确定（通常包括：batch size，如果 mini-batch 之间数据可变，也会包括图片的宽度和高度等），在定义模型

息，当遇到无法确定的维度时，相应维度指定为 None ，如下面的代码片段所示：

```
import paddle.fluid as fluid

#定义x的维度为[3,None]，其中我们只能确定x的第一的维度为3，第二个维度未知，要在程序执行过程中才能确定
x = fluid.layers.data(name="x", shape=[3,None], dtype="int64")

#batch size无需显示指定，框架会自动补充第0维为batch size，并在运行时填充正确数值
a = fluid.layers.data(name="a",shape=[3,4],dtype='int64')

#若图片的宽度和高度在运行时可变，将宽度和高度定义为None。
#shape的三个维度含义分别是：channel、图片的宽度、图片的高度
b = fluid.layers.data(name="image",shape=[3,None,None],dtype="float32")
```

其中，dtype="int64"表示有符号64位整数数据类型，更多Fluid目前支持的数据类型请查看：Fluid目前支持的数据类型。

3. **常量 Tensor**

Fluid 通过 `fluid.layers.fill_constant` 来实现常量Tensor，用户可以指定Tensor的形状，数据类型和常量值。代码实现如下所示：

```
import paddle.fluid as fluid
data = fluid.layers.fill_constant(shape=[1], value=0, dtype='int64')
```

需要注意的是，上述定义的tensor并不具有值，它们仅表示将要执行的操作，如您直接打印data将会得到描述该data的一段信息：

```
print data
```

输出结果:

```
name: "fill_constant_0.tmp_0"
type {
    type: LOD_TENSOR
    lod_tensor {
        tensor {
            data_type: INT64
            dims: 1
        }
    }
}
persistable: false
```

具体输出数值将在Executor运行时得到，详细过程会在后文展开描述。

数据传入

Fluid有特定的数据传入方式：

您需要使用 `fluid.layers.data` 配置数据输入层，并在 `fluid.Executor` 或 `fluid.ParallelExecutor` 中，使用 executor.run(feed=...) 传入训练数据。

具体的数据准备过程，请阅读准备数据

使用Operator表示对数据的操作

在Fluid中，所有对数据的操作都由Operator表示，您可以使用内置指令来描述他们的神经网络。

为了便于用户使用，在Python端，Fluid中的Operator被一步封装入 `paddle.fluid.layers` ， `paddle.fluid.nets` 等模块。

这是因为一些常见的对Tensor的操作可能是由更多基础操作构成，为了提高使用的便利性，框架内部对基础 Operator 进行了一些封装，包括创建 Operator 依赖可学习参数，可学习参数的初始化细节等，减少用户重复开发的成本。

例如用户可以利用 `paddle.fluid.layers.elementwise_add()` 实现两个输入Tensor的加法运算：

```
a = fluid.layers.data(name="a",shape=[1],dtype='float32')
b = fluid.layers.data(name="b",shape=[1],dtype='float32')

result = fluid.layers.elementwise_add(a,b)

#定义Exector
cpu = fluid.core.CPUPlace() #定义运算场所，这里选择在CPU下训练
exe = fluid.Executor(cpu) #创建执行器
exe.run(fluid.default_startup_program()) #网络参数初始化

#准备数据
import numpy
data_1 = int(input("Please enter an integer: a="))
data_2 = int(input("Please enter an integer: b="))
x = numpy.array([[data_1]])
y = numpy.array([[data_2]])

#执行计算
outs = exe.run(
feed={'a':x,'b':y},
fetch_list=[result.name])

#验证结果
print "%d+%d=%d" % (data_1,data_2,outs[0][0])
```

输出结果：

```
a=7
b=3
7+3=10
```

本次运行时，输入a=7，b=3，得到outs=10。

您可以复制这段代码在本地执行，根据指示输入其他数值观察计算结果。

如果想获取网络执行过程中的a，b的具体值，可以将希望查看的变量添加在fetch_list中。

```
...
#执行计算
outs = exe.run(
    feed={'a':x,'b':y},
    fetch_list=[a,b,result.name])
#查看输出结果
print outs
```

输出结果：

```
[array([[7]]), array([[3]]), array([[10]])]
```

使用Program描述神经网络模型

Fluid不同于其他大部分深度学习框架，去掉了静态计算图的概念，代之以Program的形式动态描述计算过程。这种动态的计算描述方式，兼具网络结构修改的灵活性和模型搭建的便捷性，在保证性能的同时极大地提高了框架对模型的表达能力。

开发者的所有 Operator 都将写入 Program ，在Fluid内部将自动转化为一种叫作 ProgramDesc 的描述语言，Program 的定义过程就像在写一段通用程序，有开发经验的用户在使用 Fluid 时，会很自然的将自己的知识迁移过来。

其中，Fluid通过提供顺序、分支和循环三种执行结构的支持，让用户可以通过组合描述任意复杂的模型。

顺序执行：

用户可以使用顺序执行的方式搭建网络：

```
x = fluid.layers.data(name='x',shape=[13], dtype='float32')
y_predict = fluid.layers.fc(input=x, size=1, act=None)
y = fluid.layers.data(name='y', shape=[1], dtype='float32')
cost = fluid.layers.square_error_cost(input=y_predict, label=y)
```

条件分支——switch、if else：

```
lr = fluid.layers.tensor.create_global_var(
        shape=[1],
        value=0.0,
        dtype='float32',
        persistable=True,
        name="learning_rate")

one_var = fluid.layers.fill_constant(
        shape=[1], dtype='float32', value=1.0)
two_var = fluid.layers.fill_constant(
        shape=[1], dtype='float32', value=2.0)

with fluid.layers.control_flow.Switch() as switch:
    with switch.case(global_step == zero_var):
        fluid.layers.tensor.assign(input=one_var, output=lr)
    with switch.default():
        fluid.layers.tensor.assign(input=two_var, output=lr)
```

关于 Fluid 中 Program 的详细设计思想，可以参考阅读Fluid设计思想 更多 Fluid 中的控制流，可以参考阅读API文档

使用Executor执行Program

Fluid的设计思想类似于高级编程语言C++和JAVA等。程序的执行过程被分为编译和执行两个阶段。

用户完成对 Program 的定义后，Executor 接受这段 Program 并转化为C++后端真正可执行的 FluidProgram，这一自动完成的过程叫做编译。

编译过后需要 Executor 来执行这段编译好的 FluidProgram。

例如上文实现的加法运算，当构建好 Program 后，需要创建 Executor，进行初始化 Program 和训练 Program：

```
#定义Exector
cpu = fluid.core.CPUPlace() #定义运算场所，这里选择在CPU下训练
exe = fluid.Executor(cpu) #创建执行器
exe.run(fluid.default_startup_program()) #用来进行初始化的program

#训练Program，开始计算
#feed以字典的形式定义了数据传入网络的顺序
#fetch_list定义了网络的输出
outs = exe.run(
    feed={'a':x,'b':y},
    fetch_list=[result.name])
```

代码实例

至此，您已经对Fluid核心概念有了初步认识了，不妨尝试配置一个简单的网络吧。如果感兴趣的话可以跟随本部分，完成一个非常简单的数据预测。已经掌握这部分内容的话，可以跳过本节阅读What's next。

从逻辑层面明确了输入数据格式、模型结构、损失函数以及优化算法后，需要使用 PaddlePaddle 提供的 API 及算子来实现模型逻辑。一个典型的模型主要包含4个部分，分别是：输入数据格式定义，模型前向计算逻辑，损失函数以及优化算法。

1. 问题描述

   给定一组数据 $< X, Y >$，求解出函数 f，使得 $y = f(x)$，其中X,Y均为一维张量。最终网络可以依据输入x，准确预测出$y_{predict}$。

2. 定义数据

   假设输入数据X=[1 2 3 4]，Y=[2,4,6,8]，在网络中定义：

   ```
   #定义X数值
   train_data=numpy.array([[1.0],[2.0],[3.0],[4.0]]).astype('float32')
   #定义期望预测的真实值y_true
   y_true = numpy.array([[2.0],[4.0],[6.0],[8.0]]).astype('float32')
   ```

3. 搭建网络（定义前向计算逻辑）

   接下来需要定义预测值与输入的关系，本次使用一个简单的线性回归函数进行预测：

```
#搭建全连接网络
y_predict = fluid.layers.fc(input=x,size=1,act=None)
```

这样的网络就可以进行预测了，虽然输出结果只是一组随机数，离预期结果仍相差甚远：

```
#加载库
import paddle.fluid as fluid
import numpy
#定义数据
train_data=numpy.array([[1.0],[2.0],[3.0],[4.0]]).astype('float32')
y_true = numpy.array([[2.0],[4.0],[6.0],[8.0]]).astype('float32')
#定义预测函数
x = fluid.layers.data(name="x",shape=[1],dtype='float32')
y_predict = fluid.layers.fc(input=x,size=1,act=None)
#参数初始化
cpu = fluid.core.CPUPlace()
exe = fluid.Executor(cpu)
exe.run(fluid.default_startup_program())
#开始训练
outs = exe.run(
    feed={'x':train_data},
    fetch_list=[y_predict.name])
#观察结果
print outs
```

输出结果：

```
[array([[0.74079144],
           [1.4815829 ],
           [2.2223744 ],
           [2.9631658 ]], dtype=float32)]
```

4. 添加损失函数

完成模型搭建后，如何评估预测结果的好坏呢？我们通常在设计的网络中添加损失函数，以计算真实值与预测值的差。

在本例中，损失函数采用均方差函数：

```
cost = fluid.layers.square_error_cost(input=y_predict, label=y)
avg_cost = fluid.layers.mean(cost)
```

输出一轮计算后的预测值和损失函数：

```
#加载库
import paddle.fluid as fluid
import numpy
#定义数据
train_data=numpy.array([[1.0],[2.0],[3.0],[4.0]]).astype('float32')
y_true = numpy.array([[2.0],[4.0],[6.0],[8.0]]).astype('float32')
#定义网络
x = fluid.layers.data(name="x",shape=[1],dtype='float32')
y = fluid.layers.data(name="y",shape=[1],dtype='float32')
y_predict = fluid.layers.fc(input=x,size=1,act=None)
#定义损失函数
cost = fluid.layers.square_error_cost(input=y_predict,label=y)
avg_cost = fluid.layers.mean(cost)
#参数初始化
cpu = fluid.core.CPUPlace()
exe = fluid.Executor(cpu)
exe.run(fluid.default_startup_program())
#开始训练
outs = exe.run(
    feed={'x':train_data,'y':y_true},
    fetch_list=[y_predict.name,avg_cost.name])
#观察结果
print outs
```

输出结果:

```
[array([[0.9010564],
    [1.8021128],
    [2.7031693],
    [3.6042256]], dtype=float32), array([9.057577], dtype=float32)]
```

5. 网络优化

确定损失函数后，可以通过前向计算得到损失值，然后通过链式求导法则得到参数的梯度值。

获取梯度值后需要更新参数，最简单的算法是随机梯度下降法：w=w−η·g，由 fluid.optimizer.SGD 实现：

```
sgd_optimizer = fluid.optimizer.SGD(learning_rate=0.01)
```

让我们的网络训练100次，查看结果：

```
#加载库
import paddle.fluid as fluid
import numpy
#定义数据
train_data=numpy.array([[1.0],[2.0],[3.0],[4.0]]).astype('float32')
y_true = numpy.array([[2.0],[4.0],[6.0],[8.0]]).astype('float32')
#定义网络
x = fluid.layers.data(name="x",shape=[1],dtype='float32')
y = fluid.layers.data(name="y",shape=[1],dtype='float32')
y_predict = fluid.layers.fc(input=x,size=1,act=None)
#定义损失函数
cost = fluid.layers.square_error_cost(input=y_predict,label=y)
avg_cost = fluid.layers.mean(cost)
#定义优化方法
sgd_optimizer = fluid.optimizer.SGD(learning_rate=0.01)
sgd_optimizer.minimize(avg_cost)
#参数初始化
cpu = fluid.core.CPUPlace()
exe = fluid.Executor(cpu)
exe.run(fluid.default_startup_program())
##开始训练，迭代100次
for i in range(100):
    outs = exe.run(
        feed={'x':train_data,'y':y_true},
        fetch_list=[y_predict.name,avg_cost.name])
#观察结果
print outs
```

输出结果:

```
[array([[2.2075021],
        [4.1005487],
        [5.9935956],
        [7.8866425]], dtype=float32), array([0.01651453], dtype=float32)]
```

可以看到100次迭代后，预测值已经非常接近真实值了，损失值也从初始值9.05下降到了0.01。

恭喜您！已经成功完成了第一个简单网络的搭建，想尝试线性回归的进阶版——房价预测模型，请阅读：线性回归。更多丰富的模型实例可以在模型库中找到。

What's next

如果您已经掌握了基本操作，可以进行下一阶段的学习了：

跟随这一教程将学习到如何对实际问题建模并使用fluid构建模型：配置简单的网络。

完成网络搭建后，可以开始在单机或多机上训练您的网络了，详细步骤请参考训练神经网络。

除此之外，使用文档模块根据开发者的不同背景划分了三个学习阶段：新手入门、使用指南和进阶使用。

如果您希望阅读更多场景下的应用案例，可以参考深度学习基础教程。已经具备深度学习基础知识的用户，可以从使用指南开始阅读。