算法的简单介绍

该示例使用了paddle develop版本(使用了autograd中的batch_hessian函数得到神经网络的高阶导数)。

`poisson2D.jl`文件内使用PyCall.jl调用Paddle实现了PINNs算法求解2D泊松方程。

假设泊松方程有如下形式：

$$\begin{cases} \Delta u = f(x), & x \in [0,1]^2 \\ u(x) = g(x), & x \in \partial[0,1]^2 \end{cases}$$

PINNs算法的基本思想为，用神经网络直接去近似方程的解，转化为如下的优化问题：

$$\min_{\theta} \quad \beta_1 \int_{\Omega \setminus \partial\Omega} \|\Delta \hat{u}(x;\theta) - f(x)\| dx + \beta_2 \int_{\partial\Omega} \|\hat{u}(x;\theta) - g(x)\| dx$$

在计算积分时，需要采用一定的近似方法，一种方法是按梯形公式，通过固定的格点上的值来近似：

$$\int_{\Omega \setminus \partial\Omega} \|\Delta \hat{u}(x;\theta) - f(x)\| dx \approx \sum_i \Delta x \|\Delta \hat{u}(x_i;\theta) - f(x_i)\|$$

但是网格方法在问题维度升高时，会面临维度灾难的问题。因此另外一个近似积分的方法是在每次迭代的时候，采用固定数量的Monte Carlo采样来计算积分：

$$\int_{\Omega \setminus \partial\Omega} \|\Delta \hat{u}(x;\theta) - f(x)\| dx \approx \sum_i a \|\Delta \hat{u}(x_i;\theta) - f(x_i)\|$$

在本文的代码实现中，主要采用这种方法。在NeuralPDE.jl里，提供了近似积分的不同trainning strategy，作为可选的参数：

```
discretization = PhysicsInformedNN(chain,
                                   strategy;
                                   init_params = nothing,
                                   phi = nothing,
                                   derivative = nothing,
                                   )
```

其中`GridTraining`表示网格方法，`StochasticTraining`表示MC采样。除此之外，还提供了`QuasiRandomTraining`和`QuadratureTraining`，在某些模型下，这种采样算法有更好的性能。

具体实现

下面，本文将阐述具体的实现细节。

相关库的导入

```
using PyCall
paddle = pyimport("paddle")
batch_hessian = paddle.autograd.batch_hessian
mes_loss = paddle.nn.MSELoss()
paddle.set_default_dtype("float64")
```

利用PyCall.jl来调用paddle，利用develop版本里的`batch_hessian`函数实现对$\Delta u(x)$的计算（注：对于无法计算高阶导数的情况，可以使用有限差分方法近似）。由于julia内默认的数据类型是`float64`，因此这里将paddle的默认数据类型也设置成`float64`。

采样，并转化成tensor

```
function sample_to_tensor(rhs_func::Function, bc_func::Function, batch_size,
bc_size)

    # sample uniformly from domain with size (batch_size, 2)
    x = sample_domain(batch_size)
    rhs = rhs_func.(x[:,1], x[:,2])
    rhs = reshape(rhs,batch_size,1)

    # sample uniformly from boundary with size (bc_size, 2)
    bc_x = sample_bc(bc_size)
    bc_value = bc_func.(bc_x[:,1],bc_x[:,2])
    bc_value = reshape(bc_value,4*bc_size,1)

    x = paddle.to_tensor(x)
    rhs = paddle.to_tensor(rhs)
    bc_x = paddle.to_tensor(bc_x)
    bc_value = paddle.to_tensor(bc_value)
    return (x, rhs, bc_x, bc_value)
end
```

实现了另外两个辅助函数`sample_domain(batch_size)`和`sample_bc(bc_size)`，利用julia的内置的`rand`等函数，实现在区域内和边界的一定数量的采样。

同时希望传入`rhs_func(x1,x2)`和`bc_func(x1,x2)`两个函数，方便直接计算$f(x_i)$和$g(x_i)$，并转化为paddle支持的tensor类型。

损失函数

```
# Is there a better way to get the slice of tensor with PyCall?
py"""
def get_slice(x,i):
    return x[i,:,i]
"""

function loss_func(NN, x, rhs, bc_x, bc_value)
    batch_size = x.shape[1]
    x.stop_gradient = false

    # using the auto diff, also could use finite difference instead
    d2u_dx2 = batch_hessian(NN, [x], create_graph=true)
    d2u_dx2 = paddle.reshape(d2u_dx2, shape=PyVector([2, batch_size, 2]))
    Laplace = py"get_slice"(d2u_dx2,0) + py"get_slice"(d2u_dx2,1)
    Laplace = paddle.reshape(Laplace, shape=PyVector([batch_size, 1]))

    # Poisson Equation Δu = f(x)
    loss = mes_loss(Laplace,rhs)
    x.stop_gradient = true

    # boundary condition: u(x) = g(x)
```

```
    NN_value = NN(bc_x)
    loss += mes_loss(NN_value, bc_value)

    return loss
end
```

使用`batch_hessian`函数最后得到的是一个shape=(dim，batch_size*dim)的tensor，通过reshape转化成shape=(dim，batch_size，dim)，此时对于样本x_i，切片[:, i, :]将得到对应的Hessian矩阵，只取该矩阵的迹，则得到对应的二阶导数Δu。

由于直接的多维切片操作在PyCall下并不支持，这里使用了PyCall的定义python函数的功能，利用辅助函数`get_slice`来获取切片。

给定样本点$x_i \in \Omega$和$x_j \in \partial\Omega$，损失函数可以表达为：

$$Loss(\hat{u}(x;\theta)) = \frac{1}{\mathcal{N}_\Omega}\sum_i |\Delta\hat{u}(x_i;\theta) - f(x_i)|^2 + \frac{1}{\mathcal{N}_{\partial\Omega}}\sum_j |u(x_j;\theta) - g(x_j)|^2$$

在这里，两项之间的权重也可以自行修改。

训练

```
function training(NN, opt, iterations::Int, rhs_func::Function,
bc_func::Function, batch_size::Int, bc_size::Int)
    for iter in 1:iterations
        x, rhs, bc_x, bc_value = sample_to_tensor(rhs_func, bc_func, batch_size,
bc_size)
        loss = loss_func(NN, x, rhs, bc_x, bc_value)
        println(loss.numpy()[1])
        loss.backward()
        opt.step()
        opt.clear_grad()
    end
end
```

需要使用paddle的optimizer优化器`opt`，并初始化一个paddle的神经网络`NN`，设定迭代的次数`iterations`，以及方程的函数和采样的样本数。

测试例子

这里测试的例子为：

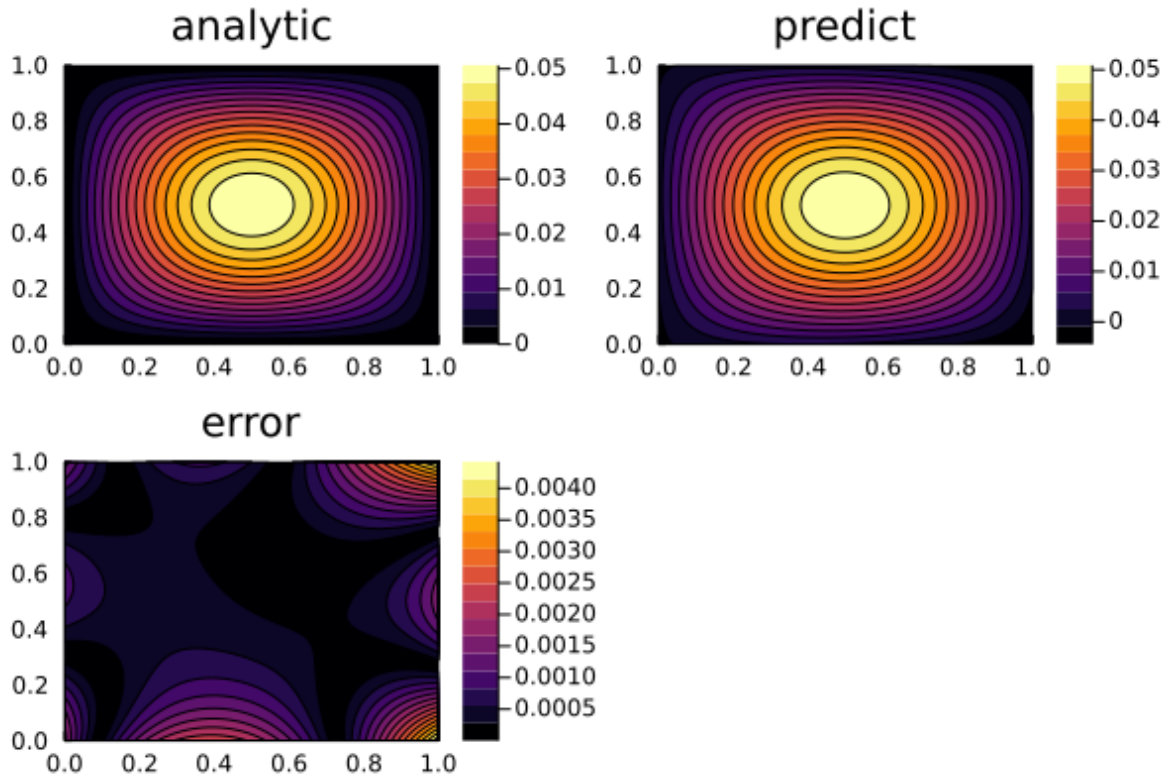
$$\begin{cases} \Delta u(x_1, x_2) = -\sin(\pi x_1)\sin(\pi x_2), & x \in [0,1]^2 \\ u(x) = 0, & x \in \partial[0,1]^2 \end{cases}$$

具体的代码在文件`paddle_demo.jl`内。使用的神经网络结构为全连接网络，三层，每层有16个神经元，激活函数为sigmoid。每次迭代中，domain内采样的样本数为100，边界上的样本数为40。使用的优化器为Adam，在前4000次迭代中学习率为0.1，在后面2000次迭代中，学习率为0.01。具体的训练代码可以写为：

```
bc_func(x1, x2) = 0.0
rhs_func(x1, x2) = -sinpi(x1)*sinpi(x2)

# initial neural network
NN = paddle.nn.Sequential(
          paddle.nn.Linear(2, 16),
```

```
            paddle.nn.Sigmoid(),
            paddle.nn.Linear(16, 16),
            paddle.nn.Sigmoid(),
            paddle.nn.Linear(16, 1)
        )

# set batch size = 100
batch_size = 100
# sample 10 points from each side of Rectangle, so sample 40 point from boundary
each iteration
bc_size = floor(Int, batch_size/10)


# initial an optimizer with lr 0.1
adam = paddle.optimizer.Adam(learning_rate=0.1,
                     parameters=NN.parameters())

# tarining, 4000 iterations with lr 0.1
training(NN, adam, 4000, rhs_func, bc_func, batch_size, bc_size)

# initial an optimizer with lr 0.1
adam = paddle.optimizer.Adam(learning_rate=0.01,
                     parameters=NN.parameters())

# tarining, 2000 iterations with lr 0.01
training(NN, adam, 2000, rhs_func, bc_func, batch_size, bc_size)
```

下图表示该方程的解析解，数值解，以及数值解的绝对值误差（由于随机性算法，误差并不会总是保持固定的水平，调整合适的学习率和优化器以及样本数，可以使误差控制在一个稳定的范围内）：



与NeuralPDE的性能比较

方程求解的精度

在相同的采样算法，优化器，神经网络结构，迭代次数，样本数的条件下，两种实现基本达到大致相同的误差范围，下表展示在五次重复实验中，两种实现各自达到的L^2误差，以及它们的平均值水平（注意，同一次实验里两种实现并不是使用同一份样本，列出五次实验仅是为了表现误差波动的大致范围，而且由于随机性算法的原因，五次实验的统计结果并不一定准确）

	实验一	**实验二**	**实验三**	**实验四**	**实验五**	**平均值**
Paddle	0.1091	0.1437	0.1059	0.0829	0.1296	0.1142
NeuralPDE	0.0785	0.1137	0.1016	0.1136	0.1463	0.1107

计算时间

利用`BenchmarkTools.jl`库的`@btime`宏估计两种实现在运行时的花费时间。其中对于上述的例子，在相同的机器上，NeuralPDE每次迭代平均花费的计算时间为0.0057(sec)，该项目内实现的算法每次迭代平均花费的计算时间为0.0074(sec)。但是NeuralPDE在第一次运行时需要较长的时间编译，相比之下paddle实现的在第一次运行时也能较快启动。