使用可微分物理梯度进行 Burgers 方程优化

目录

为了说明在可微分物理（DP）设置中计算梯度的过程，我们瞄准了与 **Burgers Optimization with a PINN** 中 PINN 示例相同的逆向问题（重构任务）。选择 DP 作为方法有一些直接的含义：我们从一个离散化的 PDE 开始，系统的演化现在完全由最终的数值求解器决定。因此，唯一真正的未知量是初始状态。我们仍然需要多次重新计算初始状态和目标状态之间的所有状态，只是现在我们不需要神经网络（NN）来完成这一步。相反，我们依赖于模型方程的离散化。

此外，当我们为 DP 方法选择初始离散化时，未知的初始状态由所涉及物理场的采样点组成，我们可以简单地将这些未知量表示为浮点变量。因此，即使是初始状态，我们也不需要设置神经网络。因此，当用 DP 求解时，我们的 Burgers 重构问题简化为一个基于梯度的优化，而不涉及任何神经网络。尽管如此，这是一个非常好的起点来说明这个过程。

(

DP 的关键区别

- **PINN 的做法**：解的形式完全由神经网络表示。我们训练网络参数，让它尽可能满足 PDE 残差和数据约束。
- **DP 的做法**：解的形式直接由 PDE 的数值解法给出。神经网络不参与 PDE 的演化。唯一需要优化的，是 **初始状态** 本身。

换句话说，DP 方法“把神经网络拿掉了”，只保留了 PDE 求解器和梯度优化。

)

首先，我们将建立我们的离散化模拟。这里我们采用 phiflow，如 Burgers 前向模拟概述部分所示。[在 colab 中运行]

[返回顶部]

初始化

phiflow 直接为我们提供了一系列可微分操作，前提是我们不使用 numpy 后端。这里重要的步骤是包含 `phi.tf.flow` 而不是 `phi.flow`（对于 pytorch，你可以使用 `phi.torch.flow`）。

因此，作为第一步，让我们设置一些常量，并用零初始化一个速度场，以及我们在 `t = 0.5`（第 16 步）的约束，现在作为 phiflow 中的 `CenteredGrid`。两者都使用周期性边界条件（通过 `extrapolation.PERIODIC`）和空间离散化 $\Delta x = 1/128$。

下面我们验证我们的模拟场现在由 TensorFlow 支持。

(

这一步的目的就是：

1. **选好后端**（TensorFlow），保证可以做自动求导。
2. **建好网格**，包括：
    - 初始速度场（全零）；
    - 目标参考解（t=0.5 时的解）。
3. 验证数据类型，确保后续能跑 DP 优化。

)

```
!pip install --upgrade --quiet phiflow==3.4
from phi.tf.flow import *

N = 128
DX = 2/N
STEPS = 32
DT = 1/STEPS
NU = 0.01/(N*np.pi)

# 分配速度网格
velocity = CenteredGrid(0, extrapolation.PERIODIC, x=N, bounds=Box(x=(-1,1)))
# 以及一个包含参考解的网格
REFERENCE_DATA = math.tensor([0.008612174447657694, 0.02584669669548606,
0.04313635726640778, ...], 'x') # (假设有128个值的张量)
SOLUTION_T16 = CenteredGrid( REFERENCE_DATA, extrapolation.PERIODIC, x=N, bounds=Box(x=
(-1,1)))

type(velocity.values.native())
# 输出: tensorflow.python.framework.ops.EagerTensor
```

梯度计算

Phiflow 的 `math.gradient` 操作为标量损失生成一个梯度函数，我们在下面使用它来计算具有选定 32 个时间步长的整个模拟的梯度。

为了将其用于 Burgers 情况，我们需要计算一个合适的损失：我们希望 $t = 0.5$ 时的解与参考数据匹配。因此，我们简单地计算第 16 步与我们的约束数组之间的 L_2 差作为损失 `loss`。然后，我们评估初始速度状态 `velocity` 相对于此损失的梯度函数。Phiflow 的 `math.gradient` 生成一个为每个参数返回梯度的函数，由于我们这里只有 `velocity` 一个参数，`grad[0]` 代表初始速度的梯度。

因为我们只约束时间步 16，所以在这个设置中实际上可以省略第 17 到 31 步。它们没有任何自由度，并且不受任何约束。然而，为了与之前的情况进行公平比较，我们包含了它们。

请注意，我们在这里进行了大量计算：首先是模拟的 32 个步骤，然后是从损失开始向后又计算了 16 个步骤。它们被梯度磁带记录，并用于将损失反向传播到模拟的初始状态。

毫不奇怪，因为我们从零开始，第 16 个模拟步骤也有一个大约 0.38 的显著初始误差。

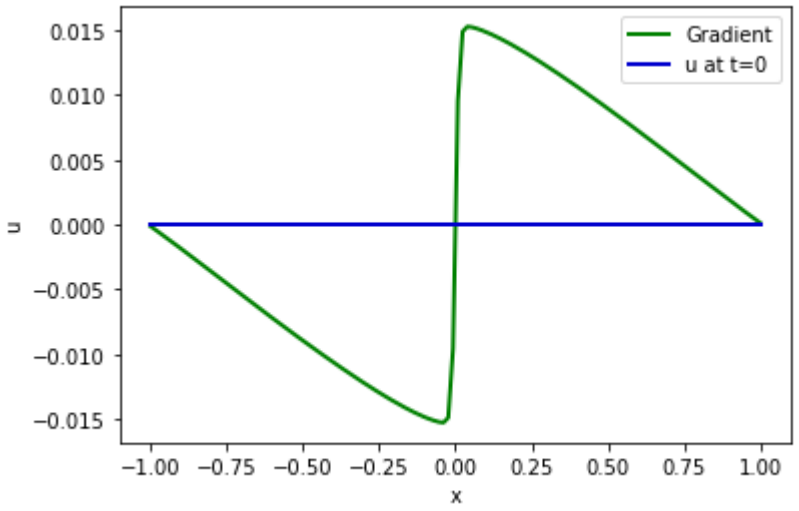
那么我们在这里得到什么梯度呢？它具有与速度相同的维度，我们可以很容易地将其可视化：从速度的零状态（蓝色显示）开始，第一个梯度在下面显示为绿线。如果你将它和解进行比较，它会指向相反的方向，正如预期的那样。解的幅度要大得多，所以我们在这里省略它（参见下一个图表）。

```python
def loss_function(velocity):
    velocities = [velocity]
    for time_step in range(STEPS):
        v1 = diffuse.explicit(1.0*velocities[-1], NU, DT, substeps=1)
        v2 = advect.semi_lagrangian(v1, v1, DT)
        velocities.append(v2)
    loss = field.l2_loss(velocities[16] - SOLUTION_T16)*2./N # MSE
    return loss, velocities

gradient_function = math.gradient(loss_function)
(loss,velocities), grad = gradient_function(velocity)
print('Loss: %f' % (loss))
# 输出: Loss: 0.382915
```

```python
import pylab as plt
fig = plt.figure().gca()
pltx = np.linspace(-1,1,N)
# 第一个梯度
fig.plot(pltx, grad[0].values.numpy('x') , lw=2, color='green', label="Gradient")
fig.plot(pltx, velocity.values.numpy('x'), lw=2, color='mediumblue', label="u at t=0")
plt.xlabel('x'); plt.ylabel('u'); plt.legend();

# 一些（可选的）其他场绘制:
# fig.plot(pltx, (velocities[16]).values.numpy('x') , lw=2, color='cyan', label="u at t=0.5")
# fig.plot(pltx, (SOLUTION_T16).values.numpy('x') , lw=2, color='red', label="solution")
# fig.plot(pltx, (velocities[16] - SOLUTION_T16).values.numpy('x') , lw=2, color='orange', label="difference")
```



这为我们提供了每个速度变量的“搜索方向”。基于线性近似，梯度告诉我们如何改变它们中的每一个以增加损失函数（梯度总是“向上”指）。因此，我们可以使用梯度来运行优化并找到最小化我们损失的初始状态 `velocity`。

(

这段代码做了一次 **完整的前向模拟**（32 步）。

定义了一个损失函数（第 16 步结果 vs 参考解）。

用自动微分算出了损失对初始速度的梯度。

)

优化过程

有了梯度，我们现在运行梯度下降优化。下面，我们使用学习率 `LR=5`，并重新评估更新状态后的损失以跟踪收敛。

在下面的代码块中，我们另外将梯度保存在一个名为 `grads` 的列表中，以便我们以后可以可视化它们。对于常规优化，我们当然可以在执行速度更新后丢弃梯度。

```
LR = 5.
grads=[]
for optim_step in range(5):
    (loss,velocities), grad = gradient_function(velocity)
    print('Optimization step %d, loss: %f' % (optim_step,loss))
    grads.append( grad[0] )
    velocity = velocity - LR * grads[-1]
# 输出:
# Optimization step 0, loss: 0.382915
# Optimization step 1, loss: 0.326882
# Optimization step 2, loss: 0.281032
# Optimization step 3, loss: 0.242804
# Optimization step 4, loss: 0.210666
```

现在我们将检查模拟的第 16 个状态在经过 5 次更新步骤后实际与目标匹配的程度。这毕竟是损失所衡量的。下一个图表以绿色显示约束（即我们想要获得的解），以及初始状态 `velocity`（我们现在通过梯度更新了五次）被求解器更新了 16 次后的重构状态。

```
fig = plt.figure().gca()
# 目标约束在 t=0.5
fig.plot(pltx, SOLUTION_T16.values.numpy('x'), lw=2, color='forestgreen',
label="Reference at t=0.5")
# 我们的模拟经过16步后的优化状态
fig.plot(pltx, velocities[16].values.numpy('x'), lw=2, color='mediumblue',
label="Simulated state at t=0.5")
plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("After 5 Optimization Steps")
```

这似乎正朝着正确的方向发展！它肯定不完美，但我们到目前为止只计算了 5 个 GD 更新步骤。激波左侧具有正速度的两个峰值和右侧的负峰值开始显现。

这是一个很好的指标，表明通过我们所有 16 个模拟步骤的梯度反向传播行为正确，并且它正在将解推向正确的方向。上面的图表只是暗示了这种设置的功能有多么强大：我们从每个模拟步骤（以及其中的每个操作）获得的梯度可以很容易地链接成更复杂的序列。在上面的例子中，我们通过所有 16 个模拟步骤进行反向传播，并且我们可以通过对代码进行微小更改来轻松扩大这种优化的“前瞻性”。

更多优化步骤

(

你的这段代码里，其实没有用到神经网络（NN），它优化的对象是 **Burgers 方程的初始状态** ——也就是初始速度场 `velocity`

)

在继续讨论更复杂的物理模拟或涉及神经网络之前，让我们先完成手头的优化任务，并运行更多步骤以获得更好的解。

回想一下**使用可微分物理梯度进行 Burgers 方程优化**中的 PINN 版本，误差在 comparable 的运行时间下更强烈地减小（大约两个数量级）。这种行为源于 DP 为整个解及其所有离散化点和时间步提供梯度，而不是局部更新。

```
import time
start = time.time()
for optim_step in range(5,50):
    (loss,velocities), grad = gradient_function(velocity)
    velocity = velocity - LR * grad[0]
    if optim_step%5==0:
        print('Optimization step %d, loss: %f' % (optim_step,loss))
end = time.time()
print("Runtime {:.2f}s".format(end-start))
# 输出:
# Optimization step 5, loss: 0.183476
# Optimization step 10, loss: 0.096224
# Optimization step 15, loss: 0.054792
# Optimization step 20, loss: 0.032819
# Optimization step 25, loss: 0.020334
# Optimization step 30, loss: 0.012852
# Optimization step 35, loss: 0.008185
# Optimization step 40, loss: 0.005186
# Optimization step 45, loss: 0.003263
# Runtime 132.33s
```

让我们再次绘制我们在 $t = 0.5$ 时的解（蓝色）与约束（绿色）的匹配程度：

```
fig = plt.figure().gca()
fig.plot(pltx, SOLUTION_T16.values.numpy('x'), lw=2, color='forestgreen',
label="Reference at t=0.5")
fig.plot(pltx, velocities[16].values.numpy('x'), lw=2, color='mediumblue',
label="Simulated state at t=0.5")
plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("After 50 Optimization Steps")
```

还不错。但是通过 16 个模拟步骤的反向传播，初始状态恢复得如何？这是我们正在改变的，因为它只是通过后来的时间观察间接约束的，所以有更多的空间偏离期望的或预期的解。

这在下一个图中显示：

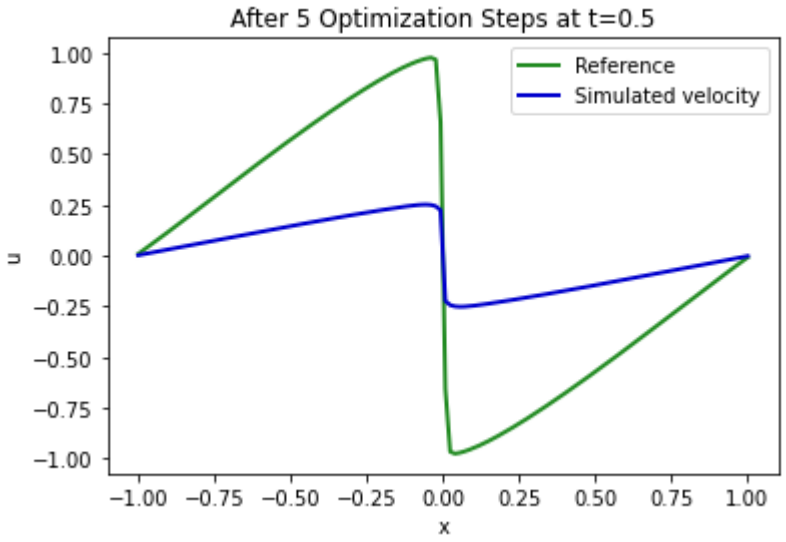
```
fig = plt.figure().gca()
pltx = np.linspace(-1,1,N)
# 时间=0时的真实状态，下移
INITIAL_GT = np.asarray( [-np.sin(np.pi * x) for x in np.linspace(-1+DX/2,1-DX/2,N)] ) # 假设初始条件是 -sin(pi*x)
fig.plot(pltx, INITIAL_GT.flatten() , lw=2, color='forestgreen', label="Ground Truth at t=0")
fig.plot(pltx, velocity.values.numpy('x'), lw=2, color='mediumblue', label="Optimized state at t=0")
plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("Initial State After 50 Optimization Steps")
```

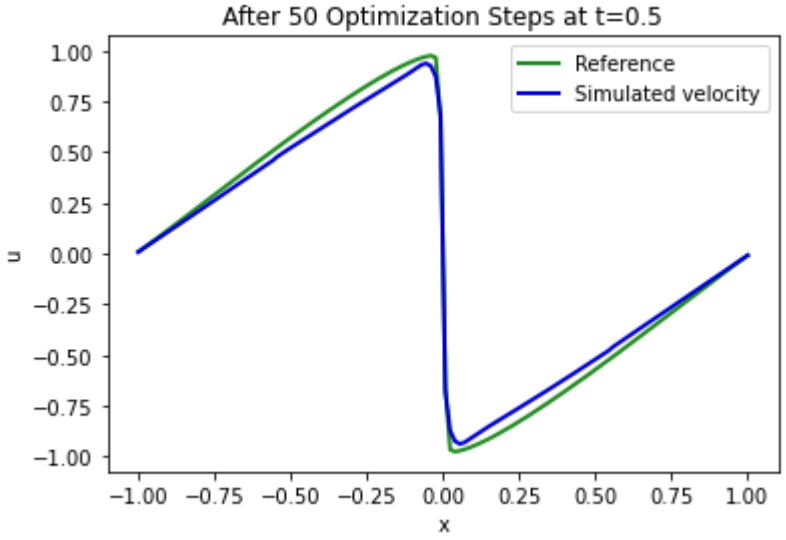
自然，这是一项更艰巨的任务：优化接收直接反馈，了解 $t = 0.5$ 时的状态应该是什么样子，但由于非线性模型方程，我们通常有大量精确或在数值上非常接近满足约束的解。因此，我们的最小化器不一定能找到我们开始时的确切状态（在默认设置下，我们可以在这里观察到来自扩散算子的一些数值振荡）。然而，在这个 Burgers 场景中，解仍然非常接近。

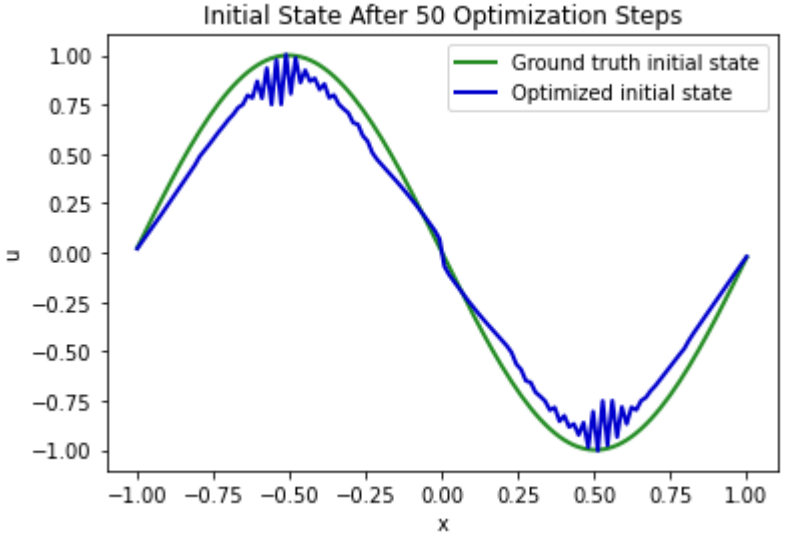
在测量重构的整体误差之前，让我们将系统随时间的完整演化可视化，因为这也会产生一个 numpy 数组形式的解，我们可以与其他版本进行比较：

```
import pylab
def show_state(a):
    a=np.expand_dims(a, axis=2)
    for i in range(4):
        a = np.concatenate( [a,a] , axis=2)
    a = np.reshape( a, [a.shape[0],a.shape[1]*a.shape[2]] )
    fig, axes = pylab.subplots(1, 1, figsize=(16, 5))
    im = axes.imshow(a, origin='upper', cmap='inferno')
    pylab.colorbar(im)

# 获取所有状态的 numpy 版本
vels = [ x.values.numpy('x,vector') for x in velocities]
# 沿向量/特征维度连接
vels = np.concatenate(vels, axis=-1)
# 保存以备与其他方法比较
import os; os.makedirs("./temp",exist_ok=True)
np.savez_compressed("./temp/burgers-diffphys-solution.npz", np.reshape(vels, [N,STEPS+1]))
show_state(vels)
```







(

这段代码完整演示了：

1. **梯度下降优化**初始速度场，使得模拟在 $t = 0.5$ 时逼近参考解。

2. **结果对比**表明末态重构效果不错，但初态不一定完全恢复。
3. **时空演化可视化**提供了更直观的结果展示，并把数据存档以便后续对比。

)

物理信息与可微分物理重构的比较

现在我们有两个版本，一个是 PINN 的，一个是 DP 的，让我们更详细地比较这两个重构。（注意：以下单元格期望之前的 Burgers-forward 和 PINN 笔记本在同一环境中执行，这样 `./temp` 目录中的 `.npz` 文件可用。）

首先让我们并排查看这些解。下面的代码生成一个包含 3 个版本的图像，从上到下：常规前向模拟给出的“真实情况”（GT）解，中间是 PINN 重构，底部是可微分物理版本。

```
# 注意，这需要之前在同一环境中运行过前向模拟和 PINN 笔记本
sol_gt=npfile=np.load("./temp/burgers-groundtruth-solution.npz")["arr_0"]
sol_pi=npfile=np.load("./temp/burgers-pinn-solution.npz")["arr_0"]
sol_dp=npfile=np.load("./temp/burgers-diffphys-solution.npz")["arr_0"]
divider = np.ones([10,33])*-1. # 我们将插入一个 -1 的块来显示黑色分隔线
sbs = np.concatenate( [sol_gt, divider, sol_pi, divider, sol_dp], axis=0)
print("\nSolutions Ground Truth (top), PINN (middle) , DiffPhys (bottom):")
show_state(np.reshape(sbs,[N*3+20,33,1]))
```

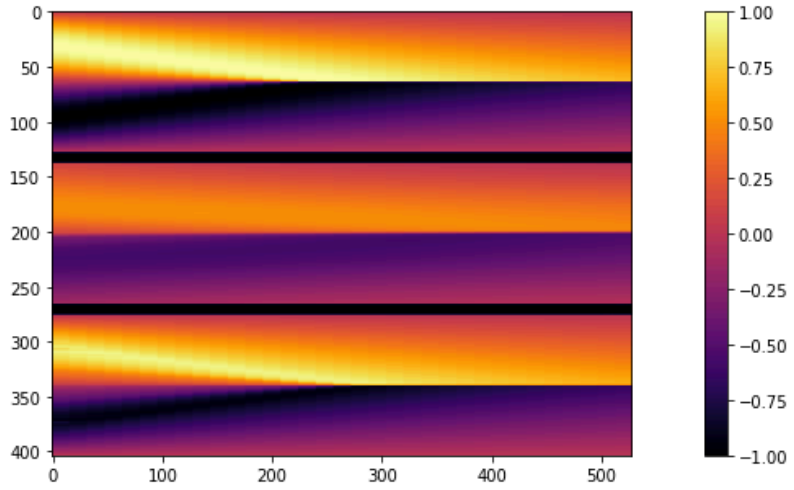
在这里可以清楚地看到，PINN 解（中间）恢复了解的总体形状，因此时间约束至少部分得到了满足。然而，它不能很好地捕捉 GT 解的幅度。

得益于在整个序列过程中改进的梯度流，使用可微分求解器进行优化（底部）的重构更接近真实情况。此外，它可以利用基于网格的离散化进行前向和后向传递，从而为未知的初始状态提供更准确的信号。尽管如此，仍然可见重构缺乏 GT 版本的某些“更尖锐”的特征，例如，在解图像的左下角可见。

让我们量化整个序列的这些误差：

```
err_pi = np.sum( np.abs(sol_pi-sol_gt)) / (STEPS*N)
err_dp = np.sum( np.abs(sol_dp-sol_gt)) / (STEPS*N)
print("MAE PINN: {:7.5f} \nMAE DP: {:7.5f}".format(err_pi,err_dp))
print("\nError GT to PINN (top) , GT to DiffPhys (bottom):")
show_state(np.reshape( np.concatenate([sol_pi-sol_gt, divider, sol_dp-sol_gt],axis=0),
[N*2+10,33,1]))
# 输出:
# MAE PINN: 0.19298
# MAE DP: 0.06382
```

```
Solutions Ground Truth (top), PINN (middle) , DiffPhys (bottom):
```



这是一个非常清楚的结果：PINN 误差比可微分物理（DP）重构的误差大 3 倍以上。

这种差异在底部联合可视化的图像中也清晰显示：DP 重构的误差幅度更接近零，如上方的紫色所示。

像这样的简单直接重构问题始终是 DP 求解器的一个很好的初始测试。在继续更复杂的设置（例如，将其与神经网络耦合）之前，可以独立测试它。如果直接优化不收敛，可能仍然存在一些根本性的错误，并且涉及神经网络也没有意义。

现在我们有了第一个例子来展示两种方法的异同。在下一节中，我们将在进入下一章更复杂的案例之前，对目前的发现进行讨论。

后续步骤

和以前一样，使用上面的代码可以改进和试验很多事情：

- 您可以尝试调整训练参数以进一步改进重构。
- 激活不同的优化器，并观察变化的（不一定是改进的）收敛行为。
- 改变步骤数，或模拟和重构的分辨率。
- 尝试在 `loss_function` 之前的一行添加 `@jit_compile`。这将包含一次性的编译成本，但会大大加快优化速度。