1 论文思路整理

1.1 提出问题

1.1.1 问题背景

随着机器学习神经网络的不断发展，人们开始探索其在微分方程求解上的应用, 其中有的研究是希望使用神经网络来代替传统方法进行求解。而这篇论文并没有完全抛弃传统方法而是希望使用神经网络来优化传统方法的求解。

1.1.2 优化程度的评判标准

假设 r_t 代表细网格上的初始状态，s_t 代表 r_t 通过下采样得到的在粗网格上的初始状态。则由这两个初始状态我们能获得在粗细两个网格上的解序列 $\{r_{t_n}\}$,$\{s_{t_n}\}$, 因为细网格上迭代的误差往往较小，所以将 $\{r_{t_n}\}$ 进行下采样所得到的粗网格上的新的解序列 $\{\mathrm{T}(r_{t_n})\}$（参考解）往往比s_{t_n} 更加接近精确解，而该论文所考虑的优化目标便是使用神经网络学习一个矫正函数 C, 使得若将 C 应用于以 s_t 为初始状态的粗网格的迭代过程能获得尽可能接近（L2 距离）$\{\mathrm{T}(r_{t_n})\}$ 的解序列。

1.2 解决方法

作者针对问题的目标分别提出了三种训练策略

1.2.1 NON

假设 $\{r_t\}$ 是参考解序列，则从其中一个状态 r_{t_1} 出发在粗网格上迭代一次得到新的状态 s_{t_1}, 而训练数据的输入为 s_{t_1}, 输出为 c_{t_1}，每一次训练的误差为 $||c_{t_1} - (r_{t_1+1} - s_{t_1})||$。

1.2.2 PRE

PRE 是一个预计算模型, 它使用约束优化问题的求解方法得到一系列的校正值。对于 s_{t_1} 与 r_{t_1+1},PRE 会得到一个希望的矫正值 c'_{t_1}(不一定是 $r_{t_1+1} - s_{t_1}$)。而训练数据的输入为 s_{t_1}, 输出为 c_{t_1}, 每一次训练的误差为 $||c_{t_1} - c'_{t_1}||$。

PRE_{SR} 则是在 PRE 的基础上保证了 c_t 的变化不会过快。

1.2.3 SOL_n

SOL_n 相较于 NON，对每个初始的 r_{t_1},SOL_n 不只是考虑迭代一步, 而是迭代并矫正之后的 n 步, 这样可以得到 $s_{t_1}, s_{t_1+1}, \cdots, s_{t_1+n}$, 假设粗网格的迭代算子是 T, 而神经网络是 C, 则 $s_{t_1+i+1} = T\left(s_{t_1+i} + C(s_{t_1+i})\right)$。此时每一次训练的误差应是每一次考虑的所有步的总误差, 所以这个训练策略需要数值迭代的过程是可微的，而作者研究这个问题的另一个目标也是借此展示他们开发的可微物理软件的优秀性能。

1.3 NS 方程的物理意义及其数值求解过程

1.3.1 物理意义

这篇论文关注的是在欧拉视角下的 NS 方程 (既守恒性方程) 的求解, 如下式所示，此时的 NS 方程由通过质量守恒定理所得到的连续性方程和通过牛顿第二定理所得到的动量方程组成:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u}) \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2}$$

其中 $\rho, \mathbf{u}, p, \mathbf{f}, \nu$ 分别代表密度，速度，压强，外力以及黏性系数。

又由不可压的假设我们可以有:

$$\nabla \cdot \mathbf{u} = 0 \tag{3}$$

1.3.2 数值求解过程

论文使用的算子分裂和投影法来求解该方程组。

投影法 由 Helmholtz 分解定理我们可以知道任何一个向量域都可以被唯一地分解成两个部分:$\mathbf{v}^* = \mathbf{v} + \nabla\phi$, 其中 $\mathbf{v}$ 是不可压的 $(\nabla \cdot \mathbf{v} = 0)$, 而 $\nabla\phi$ 是无旋的 $(\nabla \times \nabla\phi = 0)$, 所以我们可以定义一个算子 $\mathbf{P} : \mathbf{v}^* \to \mathbf{v}$, 而所谓的投影法就是将 $\mathbf{P}$ 作用于动量方程的两端, 从而可以得到:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{f}\right) \tag{4}$$

算子分裂法 算子分裂法指的是将上述方程的右端项的各个算子分开考虑，既分为四个部分：施加外力影响、施加扩散项、施加对流项和投影，假设 $\mathbf{w}_0(\mathbf{x}) := \mathbf{u}(\mathbf{x}, t), \mathbf{w}_4(\mathbf{x}) := \mathbf{u}(\mathbf{x}, t + \Delta t)$, 则整个过程可以由下图表示:

$$\mathbf{w}_0(\mathbf{x}) \overset{\text{add force}}{\longrightarrow} \mathbf{w}_1(\mathbf{x}) \overset{\text{advect}}{\longrightarrow} \mathbf{w}_2(\mathbf{x}) \overset{\text{diffuse}}{\longrightarrow} \mathbf{w}_3(\mathbf{x}) \overset{\text{project}}{\longrightarrow} \mathbf{w}_4(\mathbf{x}).$$

其中施加外力部分使用的是前向差分，施加扩散项部分考虑到稳定性使用的是后向差分，而其中对 Laplace 算子的离散直接使用的是二阶离散格式。而对于对流项来说，因为它的非线性性质, 直接使用差分方法需要时间步长足够小，否则会导致不稳定, 但时间步长足够小又会使得计算时间增大，所以作者在这里使用的是无条件稳定且容易实现的 semi-Lagrange 方法, 这个方法的原理是近似地使用了特征线法, 将 $(\mathbf{u}\cdot\nabla)\mathbf{u}$ 近似为 $\mathbf{v}\cdot\nabla\mathbf{u}$, 其中 $\mathbf{v} = \mathbf{w}_1(\mathbf{x})$ 是时间方向上的常向量, 从而将非线性转化为线性, 再使用特征线方法可以得到 $\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{x} - \Delta t \cdot \mathbf{v})$。

对于投影部分，由投影法的定义我们可以知道 $\mathbf{w}_4 = \mathbf{w}_3 - \nabla q$, 其中 $\nabla^2 q = \nabla \cdot \mathbf{w}_3$，对于这个方程的求解是先使用差分格式对 Laplace 算子和散度算子进行离散, 从而得到一个线性方程组, 再使用 CG 法进行求解就可以得到 q。

1.4 Phiflow 如何实现上述过程的可微化

Phiflow 的 math 模块将 TensorFlow、Pytorch 等框架的 tensor 封装到了一个新的类 Tensor 中, 同时也封装了 tensor 所涉及到的一些基础可微运算，所以在使用 Phiflow 时，首先需要指定 Tensor 的后端 (Pytorch 还是 TensorFlow)，再对 Tensor 的实例进行操作，而 Tensor 的一个成员变量 nativ tensor 存储了后端的 tensor, 同时也记录了 Tensor 运算过程中的一些求导的信息。

对于整个算子分裂的过程使用的各种差分格式是容易做到可微的，因为它所涉及的只是 Tensor 的基础操作如 $+, -, \times, \div$，而这些操作对于后端的 tensor 来说都是可微的。而相对难处理的是投影的过程，因为这个过程所涉及到的是使用 CG 法求解线性方程组, 虽然整个 CG 法是迭代的过程，且迭代所涉及的操作也是基础运算，可是因为迭代步较多，所涉及的基础运算也较多，此时计算保存求导信息对时间和空间都是很大消耗，所以 Phiflow 在这里并没有记录所有 CG 求解过程的基础操作的求导信息，而是将整个过程用一个包含前向传播和反向求导的函数所封装，并定制了反向求导函数。

下面是求解反向求导的思路: 假设使用 CG 法求解的线性方程组是 $\mathbf{Ax} = \mathbf{y}$, 而反向求导就是一个已知 $\frac{\partial L}{\partial \mathbf{x}}$ 希望求解 $\frac{\partial L}{\partial \mathbf{y}}$ 的问题, 而 $\frac{\partial L}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial(\mathbf{Ax})}$, 这里若假设 A 是一个可逆阵, 那么由隐函数定理我们就有 $\frac{\partial \mathbf{x}}{\partial(\mathbf{Ax})} = \left(\frac{\partial(\mathbf{Ax})}{\partial \mathbf{x}}\right)^{-1} = A^{-1}$, 从而有 $\mathbf{A}\frac{\partial L}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{x}}$, 而因为 $\frac{\partial L}{\partial \mathbf{x}}$ 是已知的, 所以我们可以再用 CG 法求解这个方程从而得到 $\frac{\partial L}{\partial \mathbf{y}}$。

1.5 训练过程的稳定性

为了保证训练的稳定性, 对于 SOL 训练策略的网络参数初值的选择不能使用随机化生成, 因为这会导致迭代的不收敛, 而对于 NON 和 PRE 训练策略来说训练过程的稳定性较高，可以使用随机的初值选择, 所以我们可以先训练得到 NON 模型, 将其作为初值训练得到 SOL_4, 再将 SOL_4 作为初值训练得到 SOL_8, 接下来依次可以得到 SOL_{16} 和 SOL_{32}。

2 无需可微迭代的训练策略 ProNON

SOL 训练策略因为每次训练都需要考虑迭代和矫正多步的误差, 所以需要迭代过程是可微的, 这样才能训练网络。某种程度上来说 NON 是只考虑一步的 SOL, 而我们可以将一次 SOL_n 的训练过程拆成 n 个 NON 的训练过程。具体来说，假设我们有一个参考解序列 $\{r_t\}$, 设 $\{q_t^0\}$=$\{r_t\}$ 对于第一步 NON 我们根据每个 q_t^0 得到经过一步的迭代状态从而构成序列 $\{s_t^0\}$, 此时进行一步 NON 的训练, 并使用此时的神经网络矫正 $\{s_t^0\}$ 得到一个矫正后的状态 $\{q_t^1\}$, 在对 $\{q_t^1\}$ 重复上述过程, 重复经过 n 次后, 重新从 $\{q_t^0\}$ 开始新的一轮训练, 这样就做到了将 SOL 拆分成若干步的 NON, 从而避免对可微迭代的依赖, 而且通过实验可以看出这种训练策略得到的模型的矫正能力强于 SOL 所得到的的模型的矫正能力。

3 结果整理

3.1 误差柱状图

Figure 1和 Figure 2分别是应用不同模型所得到的平均误差柱状图, 其中 Figure 1展示的是所有模型各自的平均误差柱状图 (所有测试数据所得到的平均误差), 而 Figure 2则是将模型应用到不同雷诺数对应的测试数据所得到的更为详细的平均误差柱状图

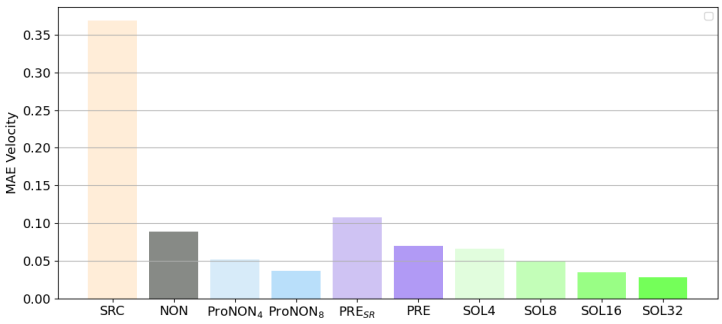

Figure 1: 不同模型对应的平均误差

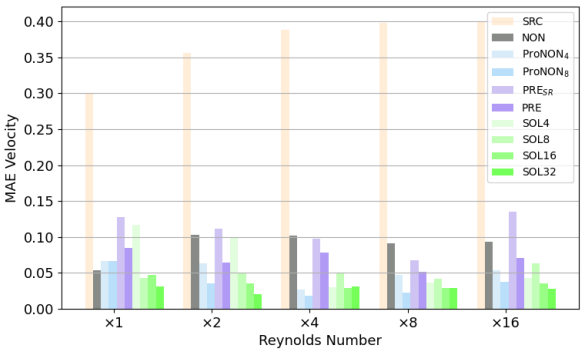

Figure 2: 不同雷诺数下不同模型对应的平均误差

从上述两幅图可以看出论文中的 NON、PRE 和 SOL 都极大的矫正了迭代过程中的误差, 其中 SOL 的效果最好, 且随着 SOL 训练时考虑的步数的增加，矫正效果也会越来越好。而通过比较 ProNON 和 SOL 也可以看出，ProNON 在和 SOL 考虑相同步数的情况下，前者有更好的矫正效果，所以其实并不需要可微的迭代器, 通过对 NON 的变形也可以达到和 SOL 一样甚至更好的效果。

3.2 两个方向上速度的热力对比图

Figure 3展示的是不使用矫正迭代和使用矫正模型 SOL_{32} 迭代所得到的不同的速度热力图，其中上半部分对应 x 方向上的速度，下半部分对应 y 方向的速度。从图中可以看出，使用矫正模型所得到的速度热力图非常接近参考图，而不使用矫正模型所得到的速度热力图与参考图偏差较大。

3.3 通用性测试

3.3.1 修改外界几何条件

将不稳定流的圆形障碍物换作方形障碍物后, 分别计算矫正迭代（SOL_{32}）与无矫正迭代的计算误差 (所有步的平均误差), 计算结果如下:

	矫正迭代	无矫正迭代
平均误差	0.25240043	0.13793774

3.3.2 修改初始速度条件

只修改了初始的速度条件后, 分别计算矫正迭代（SOL_{32}）与无矫正迭代的计算误差 (所有步的平均误差), 计算结果如下:

	矫正迭代	无矫正迭代
平均误差	0.02751089	0.3671361

3.3.3 通用性测试结论

从生成数据时所得到的的密度图可以看出, 外界几何条件会极大的影响流体的性状, 在 3.3.1 的测试中通过修改障碍物形状，会使得之前的不稳定流的摆动幅度变小很多, 而此时, 若使用之前训练好的矫正模型 SOL_{32} 不但没法正确地矫正反而会增大误差。而对于测试 3.3.2 来说，因为只是修改了初始速度，这相当于修改了雷诺数，而因为训练时考虑了雷诺数所以说此时之前的模型仍然能取得矫正的效果。

类似的，在论文中第二个例子浮力驱动模型的实验中, 作者也说了实验中训练的模型有非常好的通用性, 这其实也是因为该实验的场景没有外界几何障碍物, 所以能保持通用性。

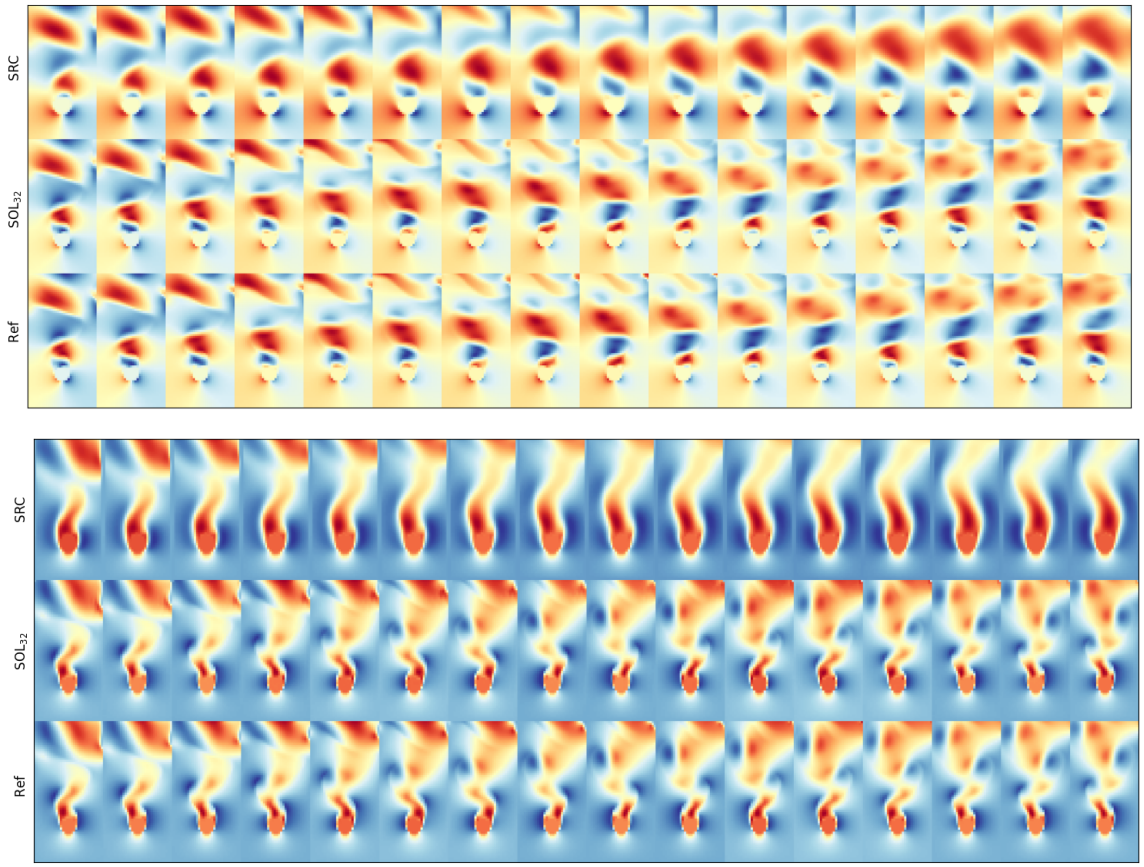

Figure 3: x 和 y 方向上的速度热力对比图

3.4 运行时间表现

对于二维不稳定流, 对于使用矫正在粗网格下迭代的平均每步的时间花费以及不使用矫正在细网格下迭代后下采样的平均每步的时间花费结果如下表所示:

	矫正迭代	无矫正迭代
CPU 时间花费 (s)	0.2512	1.3196
GPU 时间花费 (s)	0.3323	0.926

从图中可以看出 CPU 计算的矫正迭代的每一步平均时间花费大概是 GPU 计算的无矫正迭代每一步平均时间花费的 1/4。若考虑的情形是三维不稳定流，那矫正迭代的时间花费相较于无矫正迭代会更小，这是因为增加一个维度会使得细网格上的迭代时间相较于粗网格上的迭代多一个放大系数, 但神经网络的前向传播的时间花费在 GPU 计算下仍然较小。