翼型周围RANS流动的监督训练

目录

- 概述
- 公式化
- 代码即将到来...
- RANS训练数据
- 网络设置
- 训练
- 测试评估
- 下一步计划

概述

在这个监督训练的例子中，我们的目标是翼型轮廓周围的湍流气流：学习到的算子应该为给定的翼型几何形状在不同雷诺数和攻角下提供平均运动和压力分布。因此，神经网络的输入是翼型形状、雷诺数和攻角，它应该计算翼型周围具有2个分量的时间平均速度场和压力场。

这在经典上通过雷诺平均纳维-斯托克斯(RANS)模型来近似，这种设置仍然是工业中纳维-斯托克斯求解器最广泛使用的应用之一。然而，现在我们的目标不是依赖传统的数值方法来求解RANS方程，而是通过神经网络训练一个代理模型，完全绕过数值求解器。

公式化

根据监督训练的监督公式，我们的学习任务相当直接，可以写成：

$\arg\min_\theta \sum_i (f(x_i;\theta) - y_i^*)^2$

其中 x 和 y_i^* 每个都由一组物理场组成，索引 i 评估我们数据集中所有离散化点的差异。

目标是推断翼型中心周围计算域 Ω 内的速度 $u = u_x, u_y$ 和压力场 p。u_x, u_y, p 每个都具有 128^2 的维度。作为输入，我们有雷诺数 $Re \in \mathbb{R}$、攻角 $\alpha \in \mathbb{R}$ 和编码为栅格化网格的翼型形状 $s \in 128^2$。Re 和 α 根据自由流速度 f 提供，其x和y分量表示为相同大小的常数场，在翼型区域包含零。因此，总的来说，输入和输出都具有相同的维度：$x, y^* \in \mathbb{R}^{3\times128\times128}$。输入包含 $[f_x, f_y, mask]$，而输出存储通道 $[p, u_x, u_y]$。这正是我们将在下面为NN指定的输入和输出维度。

这里需要记住的一点是，我们在 y^* 中的感兴趣量包含三个不同的物理场。虽然两个速度分量在精神上相当相似，但压力场通常具有不同的行为，相对于速度有近似的平方缩放（参见伯努利方程）。这意味着我们需要小心简单的求和（如上面最小化问题中的），并且我们应该注意标准化数据。如果我们不注意，其中一个分量可能会占主导地位，而均值聚合会导致NN花费更多资源来学习大分量，而不是造成较小误差的其他分量。

代码即将到来...

让我们开始实现。请注意，我们将在这里跳过数据生成过程。下面的代码改编自[TWPH20]和这个代码库，您可以查看详细信息。在下面，我们将简单地使用在OpenFOAM中通过Spalart-Almaras RANS仿真生成的一小组训练数据。首先，让我们导入所需的模块，并从git安装数据加载器。

```
import os, sys, random
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.optim as optim

!pip install --upgrade --quiet git+https://github.com/tum-pbs/pbdl-dataset
from pbdl.torch.loader import Dataloader
```

下一个单元格将从HuggingFace下载训练数据，这可能需要一些时间...

```
BATCH_SIZE = 10
loader_train, loader_val = Dataloader.new_split(
    [320, 80],
    "airfoils",
    batch_size=BATCH_SIZE, normalize_data=None,
)
```

```
警告：`airfoils` 以单文件格式存储。下载可能需要一些时间...
成功：加载了包含400个仿真和每个1个样本的airfoils。
```

下面的PBDL数据加载器调用直接将其分为320个训练样本和80个验证样本。这些验证样本使用与训练样本相同的翼型形状，但条件不同（稍后我们将下载新形状进行测试）。

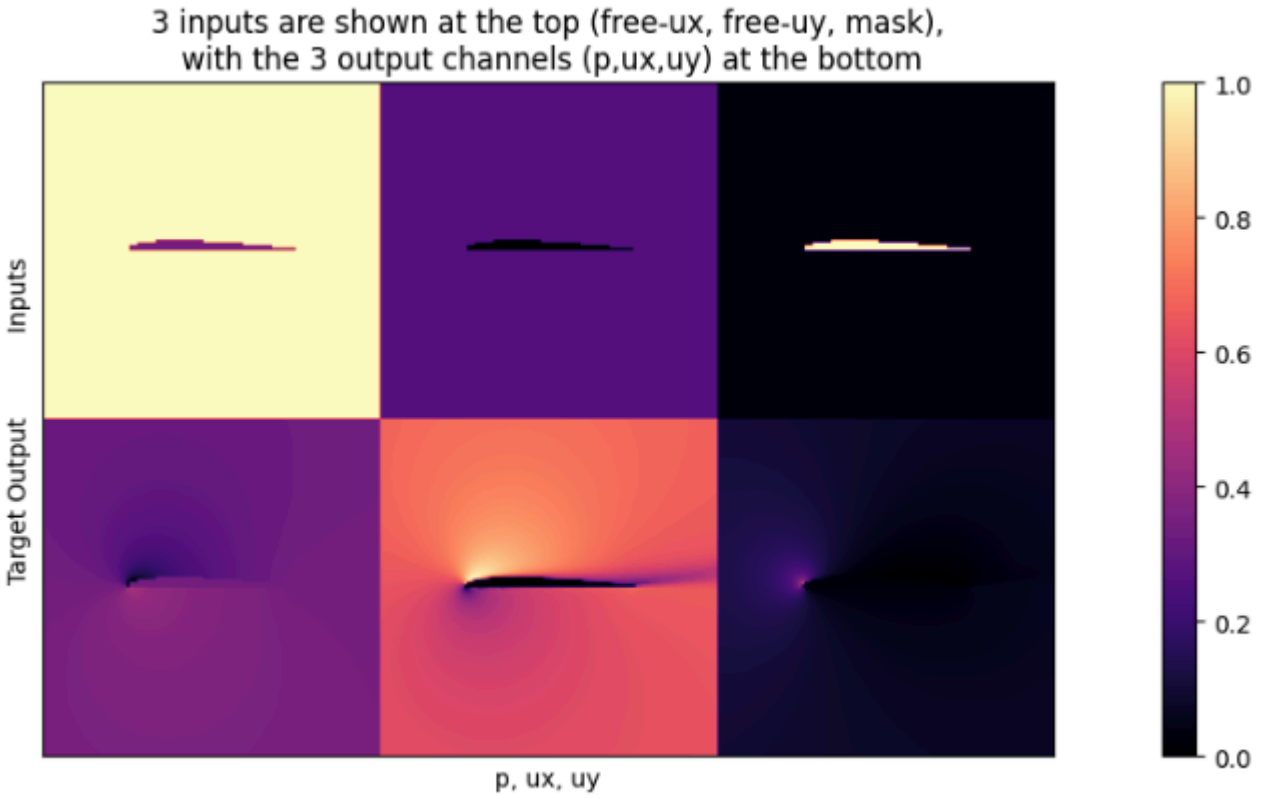
RANS训练数据

现在我们有了训练和验证数据。一般来说，尽可能多地理解我们正在使用的数据非常重要（对于任何ML任务，垃圾进垃圾出的原则绝对成立）。我们至少应该在维度和粗略统计方面理解数据，但理想情况下也应该在内容方面理解数据。否则，我们将很难解释训练运行的结果。尽管有所有AI魔法：如果你无法在数据中找出任何模式，神经网络很可能也不会找到任何有用的模式。

因此，让我们看一下其中一个训练样本。以下只是一些帮助代码来并排显示图像。

```
def plot(a1, a2, mask=None, stats=False, bottom="NN Output", top="Reference", title=None):
    c = []
    if mask is not None: mask = np.asarray(mask)
    for i in range(3):
        a2i = np.asarray(a2[i])
        if mask is not None: a2i = a2i - mask*a2i  # 可选地遮盖内部区域
        b = np.flipud(np.concatenate((a2i, a1[i]), axis=1).transpose())
        min, mean, max = np.min(b), np.mean(b), np.max(b)
        if stats:
            print("Stats %d: " % i + format([min, mean, max]))
        b -= min
        b /= max - min
        c.append(b)
    fig, axes = plt.subplots(1, 1, figsize=(16, 5))
    axes.set_xticks([]) ; axes.set_yticks([])
    im = axes.imshow(np.concatenate(c, axis=1), origin="upper", cmap="magma")
    fig.colorbar(im, ax=axes)
```

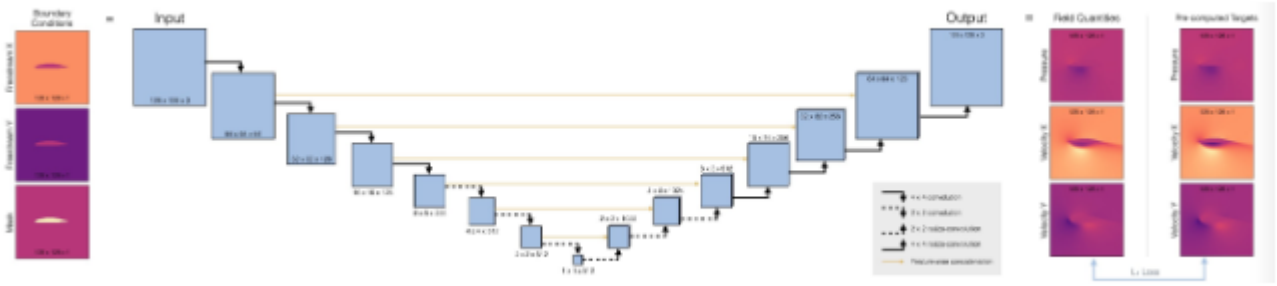
```
    axes.set_xlabel("p, ux, uy")
    axes.set_ylabel("%s %s" % (bottom, top))
    if title is not None: plt.title(title)
    plt.show()

inputs, targets = next(iter(loader_train))
plot(inputs[0], targets[0], stats=False, bottom="Target Output", top="Inputs",
title="Training sample")
```



网络设置

现在我们可以设置神经网络的架构，我们将使用一个完全卷积的U-net。这是一个广泛使用的架构，它在不同的空间分辨率上使用一堆卷积。与常规卷积网络的主要偏差是层次结构（用于全局感受野），以及引入从编码器到解码器部分的跳跃连接。这确保在特征提取过程中不会丢失信息。（请注意，这只在网络要作为一个整体使用时才有效。在我们例如希望将解码器作为独立组件使用的情况下，它不起作用。）

这是架构的概述：



首先，我们将定义一个辅助函数来设置网络中的卷积块，`blockUNet`。请注意，我们不使用任何池化！相反，我们使用步幅和转置卷积（这些需要对解码器部分对称，即具有奇数核大小），遵循最佳实践。

完整的PyTorch神经网络通过`DfpNet`类管理。

```
def blockUNet(in_c, out_c, name, size=4, pad=1, transposed=False, bn=True, activation=True,
relu=False, dropout=0.):
    block = nn.Sequential()
    if not transposed:
        block.add_module(
            "%s_conv" % name,
            nn.Conv2d(in_c, out_c, kernel_size=size, stride=2, padding=pad, bias=True)
        )
    else:
        block.add_module(
            "%s_upsam" % name, nn.Upsample(scale_factor=2, mode="bilinear")
        )
        # 为上采样（即解码器部分）减少一个核大小
        block.add_module(
            "%s_tconv" % name,
            nn.Conv2d(in_c, out_c, kernel_size=(size - 1), stride=1, padding=pad, bias=True)
        )
    if bn:
        block.add_module("%s_bn" % name, nn.BatchNorm2d(out_c))
    if dropout > 0.0:
        block.add_module("%s_dropout" % name, nn.Dropout2d(dropout, inplace=True))
    if activation:
        if relu:
            block.add_module("%s_relu" % name, nn.ReLU(inplace=True))
        else:
            block.add_module("%s_leakyrelu" % name, nn.LeakyReLU(0.2, inplace=True))
    return block

class DfpNet(nn.Module):
    def __init__(self, channelExponent=6, dropout=0.0):
        super(DfpNet, self).__init__()
        channels = int(2**channelExponent + 0.5)
        self.layer1 = blockUNet(3, channels * 1, "enc_layer1", transposed=False, bn=False,
relu=False, dropout=dropout)
        self.layer2 = blockUNet(channels, channels * 2, "enc_layer2", transposed=False,
bn=True, relu=False, dropout=dropout)
        self.layer3 = blockUNet(channels * 2, channels * 2, "enc_layer3", transposed=False,
bn=True, relu=False, dropout=dropout)
        self.layer4 = blockUNet(channels * 2, channels * 4, "enc_layer4", transposed=False,
bn=True, relu=False, dropout=dropout)
        self.layer5 = blockUNet(channels * 4, channels * 8, "enc_layer5", transposed=False,
bn=True, relu=False, dropout=dropout)
        self.layer6 = blockUNet(channels * 8, channels * 8, "enc_layer6", transposed=False,
bn=True, relu=False, dropout=dropout)
        self.layer7 = blockUNet(channels * 8, channels * 8, "enc_layer7", transposed=False,
bn=True, relu=False, dropout=dropout)

        # 注意，内核大小在解码器部分内部减少一个
```

```
        self.dlayer7 = blockUNet(channels * 8, channels * 8, "dec_layer7", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer6 = blockUNet(channels * 16, channels * 8, "dec_layer6", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer5 = blockUNet(channels * 16, channels * 4, "dec_layer5", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer4 = blockUNet(channels * 8, channels * 2, "dec_layer4", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer3 = blockUNet(channels * 4, channels * 2, "dec_layer3", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer2 = blockUNet(channels * 4, channels, "dec_layer2", transposed=True,
bn=True, relu=True, dropout=dropout)
        self.dlayer1 = blockUNet(channels * 2, 3, "dec_layer1", transposed=True, bn=False,
activation=False, dropout=dropout)

    def forward(self, input):
        # 注意，这个Unet堆栈当然可以用循环来分配...
        out1 = self.layer1(input)
        out2 = self.layer2(out1)
        out3 = self.layer3(out2)
        out4 = self.layer4(out3)
        out5 = self.layer5(out4)
        out6 = self.layer6(out5)
        out7 = self.layer7(out6)
        # ... 瓶颈 ...
        dout6 = self.dlayer7(out7)
        dout6_out6 = torch.cat([dout6, out6], 1)
        dout6 = self.dlayer6(dout6_out6)
        dout6_out5 = torch.cat([dout6, out5], 1)
        dout5 = self.dlayer5(dout6_out5)
        dout5_out4 = torch.cat([dout5, out4], 1)
        dout4 = self.dlayer4(dout5_out4)
        dout4_out3 = torch.cat([dout4, out3], 1)
        dout3 = self.dlayer3(dout4_out3)
        dout3_out2 = torch.cat([dout3, out2], 1)
        dout2 = self.dlayer2(dout3_out2)
        dout2_out1 = torch.cat([dout2, out1], 1)
        dout1 = self.dlayer1(dout2_out1)
        return dout1
```

接下来，我们可以初始化`DfpNet`的实例。

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find("BatchNorm") != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

在下面，这里的`EXPO`参数控制我们Unet特征图的指数：这直接缩放网络大小（指数为4给出一个约58.5万参数的网络）。对于具有3×128^2 = 约4.9万输出的生成NN，这是一个中等大小的网络，仍然产生快速的训练时间。因此这是一个好的起点。`weights_init`函数将卷积网络初始化为合理的初始值范围，这样我们可以直接用固定的学习率训练（否则强烈推荐学习率调度）。

```
# 通道指数来控制网络大小
EXPO = 4
torch.set_default_device("cuda:0")
device = torch.get_default_device()
net = DfpNet(channelExponent=EXPO)
net.apply(weights_init)

# 需要关注的关键参数：我们有多少参数？
nn_parameters = filter(lambda p: p.requires_grad, net.parameters())
print("可训练参数：{} -> 关键！始终保持关注...".format(sum([np.prod(p.size()) for p in
nn_parameters])))

LR = 0.0002  # 学习率
loss = nn.L1Loss()
optimizer = optim.Adam(net.parameters(), lr=LR, betas=(0.5, 0.999), weight_decay=0.)
可训练参数：585027 -> 关键！始终保持关注...
```

正如打印语句中的微妙提示所示，参数计数是训练神经网络时需要关注的关键数字。很容易更改设置，得到一个有数百万参数的网络，结果可能导致训练时间的浪费（以及潜在的训练不稳定性）。参数数量绝对必须与训练数据量匹配，也应该与网络深度成比例。这三者如何相互关联是问题相关的。

训练

最后，我们可以训练神经网络。这一步可能需要一段时间，因为训练要遍历所有320个样本100次，并持续评估验证样本以跟踪当前神经网络状态的性能。

```
EPOCHS = 200  # 训练轮数
loss_hist = []
loss_hist_val = []

if os.path.isfile("dfpnet"):  # NT_DEBUG
    print("发现现有网络，加载并跳过训练")
    net.load_state_dict(torch.load("dfpnet"))
else:
    print("从头开始训练...")
    pbar = tqdm(initial=0, total=EPOCHS, ncols=96)
    for epoch in range(EPOCHS):
        # 训练
        net.train()
        loss_acc = 0
        for i, (inputs, targets) in enumerate(loader_train):
            inputs = inputs.float()
            targets = targets.float()
            net.zero_grad()
            outputs = net(inputs)
            lossL1 = loss(outputs, targets)
```

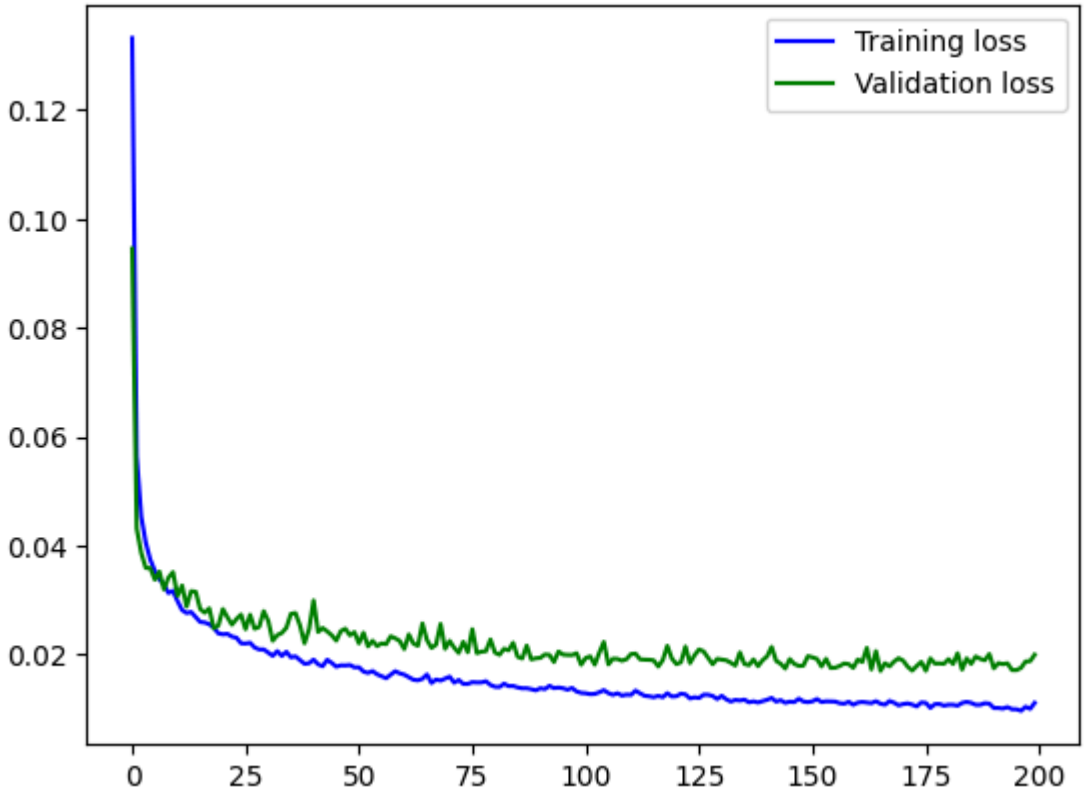
```
            lossL1.backward()
            optimizer.step()
            loss_acc += lossL1.item()
        loss_hist.append(loss_acc / len(loader_train))

        # 评估验证样本
        net.eval()
        loss_acc_v = 0
        with torch.no_grad():
            for i, (inputs, targets) in enumerate(loader_val):
                inputs = inputs.float()
                targets = targets.float()
                outputs = net(inputs)
                loss_acc_v += loss(outputs, targets).item()
        loss_hist_val.append(loss_acc_v / len(loader_val))

        pbar.set_description("训练损失：{:7.5f}，验证损失：{:7.5f}".format(loss_hist[-1],
loss_hist_val[-1]))
        pbar.update(1)

    torch.save(net.state_dict(), "dfpnet")
    print("训练完成，保存网络权重")

loss_hist = np.asarray(loss_hist)
loss_hist_val = np.asarray(loss_hist_val)
从头开始训练...
训练损失：0.01113，验证损失：0.01996：100%|██████████████████| 200/200 [05:40<00:00, 1.70s/it]
训练完成，保存网络权重
```

神经网络已训练完成，损失应该在绝对值方面有所下降：使用标准设置，验证损失从大约0.2的初始值降到训练后的约0.02。

让我们看看图表，以了解训练如何随时间进展。这通常对于识别训练中的长期趋势很重要。在实践中，很难从命令行日志中100个左右的嘈杂数字中发现整体趋势是否略微上升或下降——这在可视化中更容易发现。

您应该看到一条在约40个轮次下降的曲线，然后开始变平。在最后部分，它仍在缓慢下降，最重要的是，验证损失没有增加。这将是过拟合的确定迹象，是我们应该避免的。

```
plt.plot(np.arange(loss_hist.shape[0]), loss_hist, "b", label="训练损失")
plt.plot(np.arange(loss_hist_val.shape[0]), loss_hist_val, "g", label="验证损失")
plt.legend()
plt.show()
```



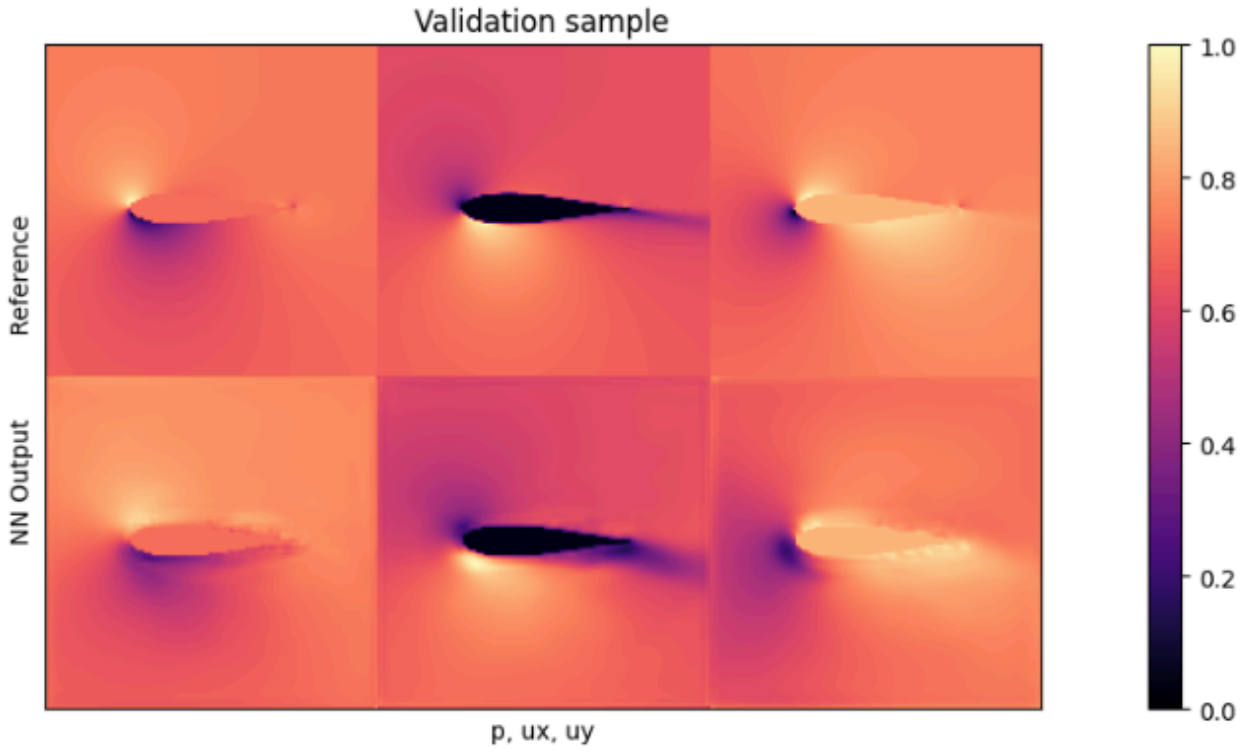
（尝试人为地减少训练数据量，然后您应该能够故意导致过拟合。）

请注意上面的验证损失通常较高，因为这里的数据集相对较小。在某些时候，网络将无法从中获得转移到验证样本的新信息。

这里有一个一般性的警告：永远不要用训练数据评估你的网络。那不会告诉你太多，因为过拟合是一个非常常见的问题。至少使用网络之前没有见过的数据，即验证数据，如果看起来不错，尝试一些更不同（至少稍微超出分布）的输入，即测试数据。下一个单元格在验证数据的一批样本上运行训练好的网络，并使用plot函数显示一个。

这里显示了输入和网络输出之间的良好相似性。翼型周围的区域通常仍然有点嘈杂（这是由狄利克雷边界引起的，可以通过修改损失和更大的网络来缓解）。压力值通常是最难学习的。不过，我们将把更详细的评估留给测试数据。

```
net.eval()
inputs, targets = next(iter(loader_val))
inputs = inputs.float()
targets = targets.float()
outputs = net(inputs)
outputs = outputs.data.cpu().numpy()
inputs = inputs.cpu()
targets = targets.cpu()
plot(targets[0], outputs[0], mask=inputs[0][2], title="验证样本")
```



测试评估

现在让我们看看实际的测试样本：在这种情况下，我们将使用新的翼型形状作为分布外(OOD)数据。这些是网络在任何训练样本中都没有见过的形状，因此它告诉我们神经网络对未见输入的泛化能力（验证数据不足以得出关于泛化的结论）。

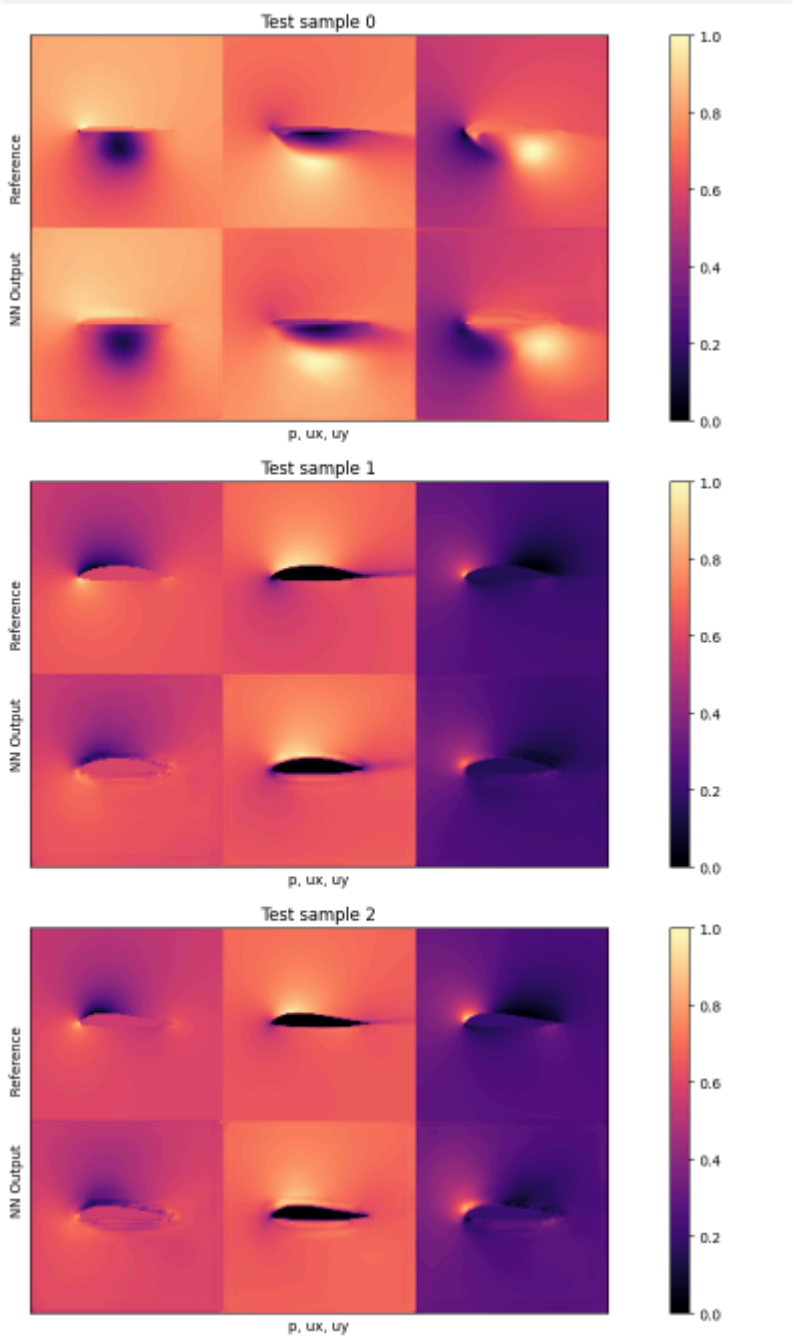
我们将使用与之前相同的可视化，正如伯努利方程所示，特别是第一列中的压力对网络来说是一个具有挑战性的量。由于它相对于输入自由流速度的三次缩放和局部峰值，它是网络推断的最困难量。

下面的单元格首先下载一个包含这些测试数据样本的较小归档文件，然后通过网络运行它们。评估循环还计算累积的L1误差，这样我们就可以量化网络在测试样本上的表现。

```
loader_test = Dataloader("airfoils-test", batch_size=1, normalize_data=None, shuffle=False)

loss = nn.L1Loss()
net.eval()
L1t_accum = 0.
for i, testdata in enumerate(loader_test, 0):
    inputs_curr, targets_curr = testdata
    inputs = inputs_curr.float()
    targets = targets_curr.float()
    outputs = net(inputs)

    outputs_curr = outputs.data.cpu().numpy()
    inputs_curr = inputs_curr.cpu()
    targets_curr = targets_curr.cpu()

    L1t_accum += loss(outputs, targets).item()
    if i<3: plot(targets_curr[0], outputs_curr[0], mask=inputs_curr[0][2], title="测试样本
%d" % i)
```

```
print("\n平均相对测试误差：{}".format(L1t_accum/len(loader_test)))
```

```
警告：`airfoils-test` 以单文件格式存储。下载可能需要一些时间...
成功：加载了包含10个仿真和每个1个样本的airfoils-test。

平均相对测试误差：0.026288176793605088
```



使用默认设置的平均测试误差应该接近0.025。由于输入已标准化，这意味着所有三个场的平均相对误差相对于每个量的最大值约为2.5%。对于新形状来说这还不错，但显然还有改进空间。

查看可视化，您会注意到特别是高压峰值和更大y速度的区域在输出中缺失。这主要是由小网络造成的，它没有足够的资源来重建细节。L_2损失也有平均行为，倾向于较大的结构（周围环境）而不是局部峰值。

尽管如此，我们已经成功地用一个小而快的神经网络架构替换了一个相当复杂的RANS求解器。它具有"开箱即用"的GPU支持（通过pytorch），是可微分的，并且只引入了几个百分点的误差。通过额外的更改和更多数据，这个设置可以变得高度准确[CT22]。

下一步计划

这里有许多显而易见的尝试（见下面的建议），例如更长的训练、更大的数据集、更大的网络等。

- 实验学习率、dropout和网络大小以减少测试集上的误差。给定训练数据，你能让它变得多小?
- 上面的设置使用标准化数据。相反，您可以通过撤销标准化来恢复原始场，以检查网络相对于原始量的表现。

正如您将看到的，您可以从这个数据集中得到的东西有点有限，前往此项目的主github仓库下载更大的数据集，或生成自己的数据。