

深度学习（Mscale 网络）求解椭圆方程代码模板 (Deep-Ritz 方法)

sis-flag

2020 年 10 月 21 日

根据神经网络的 F-principle, 网络在训练的过程中会优先拟合训练数据的低频部分。但是在某些特殊的场合下（如使用神经网络求解微分方程），需要拟合目标函数的高频部分。实验希望找到一种特殊的网络结构，更快地拟合目标函数的高频部分。

这里提出了一种新的网络结构：Mscale，用于求解椭圆方程。

椭圆方程的 Ritz 方法

椭圆方程的一般形式为

$$\begin{aligned} -\nabla(a(x)\nabla u(x)) + c(x)u(x) &= f(x) \quad x \in \Omega \\ u(x) &= u_0(x) \quad x \in \partial\Omega \end{aligned}$$

通过数学上的推导，方程可以转化为泛函的极小值问题

$$\begin{aligned} \min_u J(u) &= \int_{\Omega} \frac{1}{2}a(x)|\nabla u(x)|^2 + c(x)|u(x)|^2 - f(x)u(x) \, dx \\ \text{s.t.} \quad u(x) &= u_0(x) \quad x \in \partial\Omega \end{aligned}$$

这是一个约束优化问题，我们用加惩罚项的方法把它近似转化成无约束优化问题

$$\min_u \int_{\Omega} \frac{1}{2}a(x)|\nabla u(x)|^2 + \frac{1}{2}c(x)|u(x)|^2 - f(x)u(x) \, dx + \int_{\partial\Omega} |u(x) - u_0(x)|^2 \, dx$$

用神经网络 $N(x; \theta)$ 表示未知函数，用蒙特卡洛方法计算积分的值，就得到了最终的优化问题

$$\begin{aligned} \min_{\theta} L(\theta) = & \frac{|\Omega|}{|S_1|} \sum_{x_i \in S_1} \frac{1}{2} a(x_i) |\nabla N(x_i; \theta)|^2 + \frac{1}{2} c(x_i) |N(x_i; \theta)|^2 - f(x_i) N(x_i; \theta) \\ & + \beta \frac{|\partial\Omega|}{|S_2|} \sum_{x_i \in S_2} |N(x_i; \theta) - u_0(x_i)|^2 dx \end{aligned}$$

其中 S_1 和 S_2 是在 Ω 和 $\partial\Omega$ 上随机生成的点。

神经网络结构

一般的神经网络定义为一个函数 $N(x) : \mathbb{R}^d \rightarrow \mathbb{R}$ ，由线性变换和非线性变换复合而成。

$$N(x; \theta) = W^{[L]} \sigma(\dots W^{[2]} \sigma(W^{[1]} x + b^{[1]}) + b^{[2]} \dots) + b^{[L]}$$

其中 $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ 叫做“激活函数”，它作用在向量上相当于作用于每个分量。 L 叫做“层数”， $W^{[l]}, b^{[l]}$ 是待定的参数。

其中 $n^{[l]}$ 叫做每一层的“宽度”，满足 $n^{[1]} = d, n^{[L+1]} = 1$

$$W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}, \quad b^{[l]} \in \mathbb{R}^{n^{[l+1]}}$$

Mscale 网络结构

网络结构如图1。具体代码实现见下文。

代码实现细节

代码文件说明：

- **ritzN.py**: 用一般的 Normal 网络求解椭圆方程。
- **ritzM.py**: 用 Mscale 网络求解椭圆方程。
- **plot_result.py**: 展示网络结果，画图。
- **my_act.py**: 文件中定义了可能用到的激活函数，如果要用到新的激活函数可以在这里添加。**act_dict** 变量下标是激活函数的“名字”，它的值是名字对应的函数句柄。

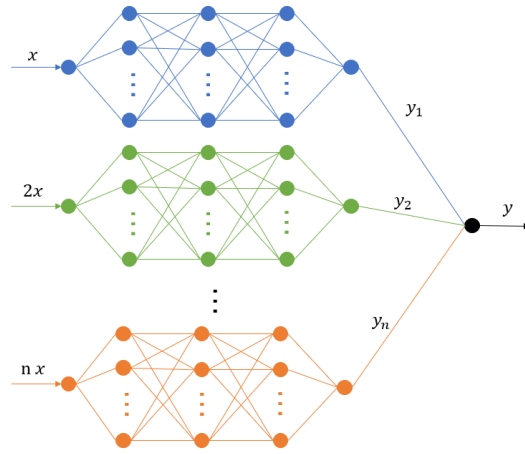


图 1: Mscale 网络结构

参数设定

在代码中，首先我们要设定所有的参数，以便修改。所有的参数和运行结果都记录在变量 **R** 中。

```
R = {}

R["seed"] = 0

R["size_it"] = 1000
R["size_bd"] = 100

R["penalty"] = 1e3

R["act_name"] = ("sin",) * 3
R["hidden_units"] = (180,) * 3
R["learning_rate"] = 1e-4
R["lr_decay"] = 5e-7

R["varcoe"] = 0.5

R["total_step"] = 5000
R["resamp_epoch"] = 1
R["plot_epoch"] = 500

R["record_path"] = os.path.join("..", "exp", "sin_N")
```

参数意义如下：

- **seed**: 随机数种子。保证两次运行代码的结果完全一样。(在必要的时候, 这个也是可调的参数之一)
- **size_it**: 区域 Ω 内部随机生成的点的个数, 也就是 $|S_1|$ 。
- **size_bd**: 区域边界 $\partial\Omega$ 上随机生成的点的个数, 也就是 $|S_2|$ 。
- **penalty**: 惩罚系数, 就是公式中的 β 。
- **scale**: Mscale 网络中的变换系数。(Normal 网络中没有这个参数)
- **act_name**: 长度为 N 的列表, 代表每一层的激活函数。
- **hidden_units**: 长度为 N 的列表, 代表每一层的宽度。
- **learning_rate**: 学习率。(一般不能太大)
- **lr_decay**: 学习率下降速度。
- **varcoe**: 初始化参数。
- **total_step**: 训练总步数。
- **resamp_epoch**: 采样间隔。每隔 X 步重新随机生成 S_1, S_2 中的点。 $X=1$ 代表每计算一步都重新随机生成点。 X 大于总步数时, 代表训练用的样本不变。
- **plot_epoch**: 从第 0 步开始, 每隔 X 步记录下数值解图像。同时输出当前状态。
- **record_path**: 保存记录的文件夹路径。在运行结束的时候, 源代码文件会被复制到文件夹下的 **code.py** 文件中, 以便重复实验。数据结果和参数会保存在文件夹下的 **data.pkl** 文件中, 以便画图和展示。

定义求解区域

这一部分定义了求解区域 Ω 。如果要求解复杂区域上的方程, 就要修改这部分代码。

```
R["dimension"] = dim = 2
R["area_it"] = 4
R["area_bd"] = 8

# interior data
def rand_it(size):
    x_it = np.random.rand(size, dim) * 2 - 1
    return x_it.astype(np.float32)
```

```
# boundary data
def rand_bd(size):

    x_bd = np.random.rand(size, dim) * 2 - 1
    ind = np.random.randint(dim, size=size)
    x01 = np.random.randint(2, size=size) * 2 - 1
    for ii in range(size):
        x_bd[ii, ind[ii]] = x01[ii]
    return x_bd.astype(np.float32)
```

这里 **dim** 是方程的空间维数，**area_it** 表示 Ω 的面积。**area_bd** 表示 $\partial\Omega$ 的面积。

rand_it 函数，生成 Ω 内均匀分布的随机点，输入需要的随机点个数 **size**，返回一个大小为 $size \times dim$ 的数组。

rand_bd 函数，生成边界 $\partial\Omega$ 上均匀分布的随机点，输入输出同上。

这样就定义了求解区域。这里对应的求解区域是 $[-1, 1]^2$ 。想要在其它区域上求解，只要重新实现一遍这个函数。

定义方程

这一部分定义了方程中的已知函数 $a(x), c(x), f(x)$ ，和精确解（边界条件） $u(x)$ 。如果精确解未知， $u(x)$ 就代表边界条件。如果要求解其它方程，就要修改这部分代码。

```
# PDE problem
# - (a(x) u'(x))' + c(x) u(x) = f(x)
R["mu"] = mu = 6 * np.pi

def u(x):
    u = np.sin(mu * x)
    return u.astype(np.float32).reshape((-1, 1))

def a(x):
    a = np.ones((x.shape[0], 1))
    return a.astype(np.float32).reshape((-1, 1))

def c(x):
    c = np.zeros((x.shape[0], 1))
    return c.astype(np.float32).reshape((-1, 1))
```

```
def f(x):
    f = mu * mu * np.sin(mu * x)
    return f.astype(np.float32).reshape((-1, 1))
```

需要注意的是：这些函数，输入都是一个大小为 $size \times dim$ 的数组，每一行代表一个采样点。返回的是大小为 $size \times 1$ 的数组。（这里的维度不一致会报错）

保存参数

把源代码文件复制到文件夹下的 **code.py** 文件中，以便重复实验。设定随机数种子。

```
# save parameters

# prepare folder
if not os.path.isdir(R["record_path"]):
    os.mkdir(R["record_path"])

# save current code
save_file = os.path.join(R["record_path"], "code.py")
shutil.copyfile(__file__, save_file)

# set seed
tf.set_random_seed(R["seed"])
tf.random.set_random_seed(R["seed"])
np.random.seed(R["seed"])
```

用于画图和计算误差的采样点

这里的 **x_test** 是用于计算误差的采样点，**u_test_true** 是这些点上的精确解的值。计算数值解和精确解误差的时候就在这些点上计算。

x_samp 是用于画图的采样点。**u_samp_true** 是这些点上的精确解的值。

如果我们不知道精确解，而是通过差分方法或者其它方法生成的参考解。这里的 **u_test_true** 就要从文件里面读取出来。

```
# get test points
x_test = np.linspace(-1, 1, 200 + 1).reshape((-1, 1))

R["x_test"] = x_test.astype(np.float32)
R["u_test_true"] = u(R["x_test"])
```

```
# get sample points
x_samp = np.linspace(-1, 1, 200 + 1).reshape((-1, 1))

R["x_samp"] = x_samp.astype(np.float32)
R["u_samp_true"] = u(R["x_samp"])
```

需要注意的是：这里的 **x_test** 和 **x_samp** 都是 $size \times dim$ 的数组，每一行代表一个采样点。这里的 **u_test_true** 和 **u_samp_true** 都是 $size \times 1$ 的数组。在画图时要另外调整。

之后在训练的过程中，还会有 **u_test_X** 和 **u_samp_X** 变量，代表训练到第 X 步的数值解。

神经网络

用 tensorflow 实现深度学习的网络结构。

如果要换网络结构或者参数初始化，就要改这段代码。

下面这段代码实现了一个普通的网络（Normal），**init_W** 和 **init_b** 是变量的初始值。

```
units = (dim,) + R["hidden_units"] + (1,)

def neural_net(x):
    with tf.variable_scope("vscope", reuse=tf.AUTO_REUSE):

        y = x

        for i in range(len(units) - 2):
            init_W = np.random.randn(units[i], units[i + 1]).astype(np.float32)
            init_W = init_W * (2 / (units[i] + units[i + 1])) ** R["varcoe"]
            init_b = np.random.randn(units[i + 1]).astype(np.float32)
            init_b = init_b * (2 / (units[i] + units[i + 1])) ** R["varcoe"]

            W = tf.get_variable(name="W" + str(i), initializer=init_W)
            b = tf.get_variable(name="b" + str(i), initializer=init_b)

            y = act_dict[R["act_name"]][i](tf.matmul(y, W) + b)

        init_W = np.random.randn(units[-2], units[-1]).astype(np.float32)
        init_W = init_W * (2 / (units[-2] + units[-1])) ** R["varcoe"]
```

```

init_b = np.random.randn(units[-1]).astype(np.float32)
init_b = init_b * (2 / (units[i] + units[i + 1])) ** R["varcoe"])

W = tf.get_variable(name="W" + str(len(units) - 1), initializer=init_W)
b = tf.get_variable(name="b" + str(len(units) - 1), initializer=init_b)

y = tf.matmul(y, W) + b

return y

```

下面这段代码实现了 Mscale 网络，原理类似。

```

units = (dim,) + R["hidden_units"] + (1,)

def neural_net(x):
    with tf.variable_scope("vscope", reuse=tf.AUTO_REUSE):

        all_y = []
        for k in range(len(R["scale"])):
            scale_y = R["scale"][k] * x

            for i in range(len(units) - 2):
                init_W = np.random.randn(units[i], units[i + 1]).astype(np.float32)
                init_W = init_W * (2 / (units[i] + units[i + 1])) ** R["varcoe"])
                init_b = np.random.randn(units[i + 1]).astype(np.float32)
                init_b = init_b * (2 / (units[i] + units[i + 1])) ** R["varcoe"])

                W = tf.get_variable(name="W" + str(i) + str(k),
                                    initializer=init_W)
                b = tf.get_variable(name="b" + str(i) + str(k),
                                    initializer=init_b)

                scale_y = act_dict[R["act_name"]][i](tf.matmul(scale_y, W) + b)

            init_W = np.random.randn(units[-2], units[-1]).astype(np.float32)
            init_W = init_W * (2 / (units[i] + units[i + 1])) ** R["varcoe"])
            init_b = np.random.randn(units[-1]).astype(np.float32)
            init_b = init_b * (2 / (units[i] + units[i + 1])) ** R["varcoe"])

            W = tf.get_variable(name="W" + str(len(units) - 1) + str(k),
                                initializer=init_W)
            b = tf.get_variable(name="b" + str(len(units) - 1) + str(k),
                                initializer=init_b)

```



```

        scale_y = tf.matmul(scale_y, W) + b

        all_y.append(scale_y)

    y = sum(all_y)

    return y

```

损失函数

定义损失函数。这里变量前面的“V”代表它的类型是 tensorflow 中的 Variable，变量前面的“N”表示它保存变量当前的取值。

如果求解的不是椭圆方程，或者损失函数的形式有变化，就要改这段代码。

```

# loss and optimizer ("V" for variable)
with tf.variable_scope("vscope", reuse=tf.AUTO_REUSE):

    Vx_it = tf.placeholder(tf.float32, shape=(None, dim))
    Vx_bd = tf.placeholder(tf.float32, shape=(None, dim))

    Va_it = tf.placeholder(tf.float32, shape=(None, 1))
    Vc_it = tf.placeholder(tf.float32, shape=(None, 1))
    Vf_it = tf.placeholder(tf.float32, shape=(None, 1))

    Vu_true_bd = tf.placeholder(tf.float32, shape=(None, 1))

    Vu_it = neural_net(Vx_it)
    Vu_bd = neural_net(Vx_bd)

    Vdu_it = tf.gradients(Vu_it, Vx_it)[0]

    Vloss_it = R["area_it"] * tf.reduce_mean(
        1 / 2 * Va_it * tf.reduce_sum(tf.square(Vdu_it), axis=1, keepdims=True)
        + 1 / 2 * Vc_it * tf.square(Vu_it)
        - Vf_it * Vu_it
    )
    Vloss_bd = R["area_bd"] * tf.reduce_mean(tf.square(Vu_bd - Vu_true_bd))

    Vloss = Vloss_it + R["penalty"] * Vloss_bd

    learning_rate = tf.placeholder_with_default(input=1e-3, shape=[], name="lr")

```

```
optimizer = tf.train.AdamOptimizer(learning_rate)
train_op = optimizer.minimize(Vloss)
```

开始训练

这里的代码一般不用大改，但是如果前面有改动，这里的要和之前的匹配。

随机生成定义域里的点，计算这些点上的函数值。

```
# generate new data ("N" for number)
if epoch % R["resamp_epoch"] == 0:
    Nx_it = rand_it(R["size_it"])
    Nx_bd = rand_bd(R["size_bd"])
    Na_it = a(Nx_it)
    Nc_it = c(Nx_it)
    Nf_it = f(Nx_it)
    Nu_bd = u(Nx_bd)
```

把生成的数据喂给网络，训练一步。

```
_, Nloss, Nloss_bd = sess.run(
    [train_op, Vloss, Vloss_bd],
    feed_dict={
        Vx_it: Nx_it,
        Vx_bd: Nx_bd,
        Va_it: Na_it,
        Vc_it: Nc_it,
        Vf_it: Nf_it,
        Vu_true_bd: Nu_bd,
        learning_rate: lr,
    },
)
```

代入采样点，计算数值解和真实解之间的误差

```
R["u_test"] = sess.run(Vu_it, feed_dict={Vx_it: R["x_test"]})
Nerror = R["area_it"] * np.mean((R["u_test"] - R["u_test_true"]) ** 2)
```

每过 `plot_epoch` 步，记录下当时的数值解，同时输出当前误差。

```
if epoch % R["plot_epoch"] == 0:
    print("epoch%d, time%.3f" % (epoch, time.time() - t0))
    print("total loss%f, boundary loss%f" % (Nloss, Nloss_bd))
    print("interior error%f" % (Nerror))
```

```
u_samp = sess.run(Vu_it, feed_dict={Vx_it: R["x_samp"]})  
R["u_samp_" + str(epoch)] = u_samp
```

最后保存所有参数

```
# save data  
data_dir = os.path.join(R["record_path"], "data.pkl")  
with open(data_dir, "wb") as file:  
    pickle.dump(R, file)
```