A note of using Tensorflow to code Laplacian operator in high dimension

Zhi-Qin John Xu*
xuzhiqin@sjtu.edu.cn
School of Mathematical Sciences,
MOE-LSC and Institute of Natural Sciences,
Shanghai Jiao Tong University,
Shanghai, 200240, P.R. China

Abstract

In this note, we give an example of using Tensorflow 1 to code Laplacian operator. We sovle a high dimensional Poisson equation for illustration. The code for Tensorflow 2 can be done similarly. Code can be found in `https://github.com/xuzhiqin1990/laplacian`

Consider the solution to the following high dimensional elliptic PDE,

$$\Delta u(\boldsymbol{x}) = g(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega, \tag{1}$$

$$u(\boldsymbol{x}) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega. \tag{2}$$

In our numerical test of this note, the loss function is chosen to be

$$L(f_\theta) = \frac{1}{n}\sum_{\boldsymbol{x}\in S} |\Delta f_\theta(\boldsymbol{x}) - g(\boldsymbol{x})|^2 + \beta\frac{1}{\tilde{n}}\sum_{\boldsymbol{x}\in\tilde{S}} (f_\theta(\boldsymbol{x}) - u_0(\boldsymbol{x}))^2. \tag{3}$$

where $f_\theta(\boldsymbol{x})$ is the DNN ouput, S is the sample set from Ω and n is the sample size. $\tilde{n}$ indicates sample set from $\partial\Omega$. The second penalty term is to enforce the boundary condition.

To see the learning accuracy, we also compute the distance between $f_\theta(\boldsymbol{x})$ and u_{true},

$$\text{MSE}(f_\theta, u_{\text{true}}) = \frac{1}{n+\tilde{n}}\sum_{\boldsymbol{x}\in S\cup\tilde{S}} (f_\theta(\boldsymbol{x}) - u_{\text{true}}(\boldsymbol{x}))^2 \tag{4}$$

A common mistake to code Laplacian operator is

```
grady=tf.gradients(y,x)
grad2y=tf.gradients(grady,x)
```

We expand the above computation as follows.

$$\text{grady} = (\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \cdots, \frac{\partial y}{\partial x_d}), \tag{5}$$

$$\text{grady} = (\sum_{i=1}^{d}\frac{\partial^2 y}{\partial x_1\partial x_i}, \sum_{i=1}^{d}\frac{\partial^2 y}{\partial x_2\partial x_i}, \cdots, \sum_{i=1}^{d}\frac{\partial^2 y}{\partial x_d\partial x_i},), \tag{6}$$

where $\boldsymbol{x} = (x_1, x_2, \cdots, x_d)$. This is because tf.gradients is designed for gradient descent, which requires the summation of the derivatives of all outputs w.r.t. the considered variable.

The difficulty of using Tensorflow to code Laplacian is as follows.

*Thank Zheng Ma and Gengjian Chen for helpful discussion. January 16, 2020.

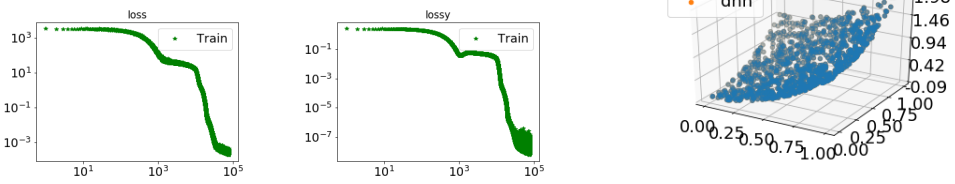
Preprint. Work in progress.

```
grady=tf.gradients(y,x)
grad2y=tf.gradients(grady[i],x[i])
```

grad2y is None in the above code. x[i] is a new variable, therefore, grady[i] has no relation with x[i] and the derivative can not be performed. A naive idea is that each dimension of input is represented by a variable. But this would be very tedious. We can use function unstack,

```
x = tf.placeholder(tf.float32, shape=[None, R['input_dim']],name="x")
#Laplacian
# To compute laplacian, we need to unstack of x, such that
# each dimension is one variable
_x=tf.unstack(x,axis=1)
#unstack each dimension, so each dim is a variable
# We pack all dim together so we can get their predict values
    # first, we need to expand each dim to shape of [-1,1]
    # then, pack them as the final shape as [-1,1]
_x_=[tf.expand_dims(tmp,axis=1) for tmp in _x]
_x2=tf.transpose(tf.stack(_x_))[0]
# univAprox2 is the DNN that returns predicted labels
y= univAprox2(_x2)
# the shape of _grady is [n_dim, x_len]
_grady=[tf.squeeze(tf.gradients(y,tmp)) for tmp in _x]
# the shape of grady is [x_len,n_dim]
grad2y=0
for ii in range(R['input_dim']):
    # take out each dimension and do grad
    grad2y+=tf.stack(tf.gradients(_grady[ii],_x_[ii]))[0]
```

Note that the completed code is at the last page. Let $u_{\text{true}} = \sum_{i=1}^{d} x_i^2$. Fig. 1 shows that the DNN can well solve the PDE well. Finally, we want to point out that it is still computational cost to perform high dimensional derivative. The alternative approach to solve such PDEs can use Ritz loss function. There could be one advantage for high dimensional derivative in the loss function. According to the F-Principle (Xu et al. 2019), i.e., the DNN captures the low frequency first, with high dimensional derivative in the loss function, the converging of high frequency component could be faster.



(a) $L(f_\theta)$ vs. epoch (b) $\text{MSE}(f_\theta, u_{\text{true}})$ vs. epoch (c) $f_\theta, u_{\text{true}}$ vs. $\boldsymbol{x}$

Figure 1

```
# Our inputs will be a batch of values taken by our functions
# x is the interior points
x = tf.placeholder(tf.float32, shape=[None, R['input_dim']],name="x")
# x_len is the length of x
x_len= tf.placeholder_with_default(input=1,shape=[],name='xlen')
# xb is the sample on boundary
xb = tf.placeholder_with_default(input=
                np.float32(np.zeros([1,R['input_dim']])),
                shape=[None, R['input_dim']], name="xb")
# y_true is the label of interior samples
y_true = tf.placeholder_with_default(input=[[0.0]],
                shape=[None, R['output_dim']], name="y")
# y_true_b is the label of the sample on boundary
y_true_b = tf.placeholder_with_default(input=[[0.0]],
                shape=[None, R['output_dim']],name="yb")
# tg2u is the true 2nd gradient of interior samples
tg2u = tf.placeholder_with_default(input=[[0.0]],
                shape=[None, R['output_dim']],name="tg2u")
# beta is the penality weight of boundary
beta= tf.placeholder_with_default(input=1.0,shape=[],name='beta')
# _lr is the learning rate of each step.
_lr= tf.placeholder_with_default(input=1e-3,shape=[],name='lr')
#Laplacian
# To compute laplacian, we need to unstack of x, such that
# each dimension is one variable
_x=tf.unstack(x,axis=1)
#unstack each dimension, so each dim is a variable
# We pack all dim together so we can get their predict values
    # first, we need to expand each dim to shape of [-1,1]
    # then, pack them as the final shape as [-1,1]
_x_=[tf.expand_dims(tmp,axis=1) for tmp in _x]
_x2=tf.transpose(tf.stack(_x_))[0]
# all inputs combines both the interior and boundary samples.
_x_0=tf.concat([_x2,xb],axis=0)
_y=tf.concat([y_true,y_true_b],axis=0)
# univAprox2 is the DNN that returns predicted labels
y= univAprox2(_x_0, R['hidden_units'],
              input_dim = R['input_dim'],
              output_dim_final=R['output_dim'])

# only consider the prediected values of
# interior points, that is, y[0:x_len].
# the shape of _grady is [n_dim, x_len]
_grady=[tf.squeeze(tf.gradients(y[0:x_len],tmp)) for tmp in _x]
# the shape of grady is [x_len,n_dim]
#grady= tf.transpose(tf.stack(_grady) )
grad2y=0
for ii in range(R['input_dim']):
    # take out each dimension and do grad
    grad2y+=tf.stack(tf.gradients(_grady[ii],_x_[ii]))[0]
loss0=tf.reduce_mean(tf.square(grad2y-tg2u))
lossb=tf.reduce_mean(tf.square(y_true_b-y[x_len:]))
loss=loss0+beta*lossb
lossy=tf.reduce_mean(tf.square(_y-y))
# We define our train operation using the Adam optimizer
adam = tf.compat.v1.train.AdamOptimizer(learning_rate=_lr)
train_op = adam.minimize(loss)
```

References

Xu, Z.-Q. J., Zhang, Y., Luo, T., Xiao, Y. & Ma, Z. (2019), 'Frequency principle: Fourier analysis sheds light on deep neural networks', *arXiv preprint arXiv:1901.06523* .