暑假作业: 重复文献 PNAS, 115, 8505-8510(2018) 中的结果

吴敏聪

更新：*August 16, 2019*

摘　　要

本文重复了文献 Solving high-dimensional partial differential equations using deep learning 中的主要结果。通过分析文献中介绍的基本思路和具体操作细节，得到了文献中类似的主要结果。在此基础上，可以分析其中的问题以及使用深度学习求解偏微分方程的启发和思路。

关键词: 深度学习，偏微分方程，tensorflow

1 文献介绍

文献的主要内容是介绍了使用基于 tensorflow 的深度学习方法求解 semilinear parabolic PDEs 的思路，同时给出了几个具体的例子。

1.1 问题

有偏微分方程

$$\frac{\partial u}{\partial t}(t,\vec{x}) + \frac{1}{2}Tr(\sigma\sigma^T(t,\vec{x})(Hess_{\vec{x}}(t,\vec{x}))) + \vec{\nabla}u(t,\vec{x})\cdot\vec{\mu}(t,\vec{x}) + f(t,\vec{x},u(t,\vec{x}),\sigma^T(t,\vec{x})\vec{\nabla}u(t,\vec{x})) = 0 \quad (1)$$

其中 σ,μ,f 均已知,σ 为 $d\times d$ 维矩阵.

并且 $\Delta\vec{u}(t,\vec{x}) = (\frac{\partial u}{\partial x_1},\frac{\partial u}{\partial x_2},\cdots,\frac{\partial u}{\partial x_d})$, $Hess_{\vec{x}}(t,\vec{x}) = \begin{pmatrix} \frac{\partial^2 u}{\partial {x_1}^2} & \cdots & \frac{\partial^2 u}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 u}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 u}{\partial {x_d}^2} \end{pmatrix}$.

边界条件 $u(T,\vec{x})$=g($\vec{x}$)，目标是求解 0 时刻给定点 $\vec{\xi}$ 的函数值 $u(0,\vec{\xi}),\vec{\xi}\in R^d$.

1.2 思路

首先，带求解的函数值 $u(0,\vec{\xi})$ 依赖于两个因素，一个是由微分方程描述的演化过程，还有一个是边界条件，在这里可以称作终止条件 $g(\vec{x})$.

暂不考虑微分演化过程对我们求解产生的影响，即姑且假设我们对微分方程的实现是完美的。那么决定待求值 $u(0,\vec{\xi})$ 的全部信息就蕴含在终止条件中.

紧接着因为我们采用的自然是数值求解的办法，不可能使用上终止条件在整个空间 R^d 上的全部取值，只能“随机”出一个子集。

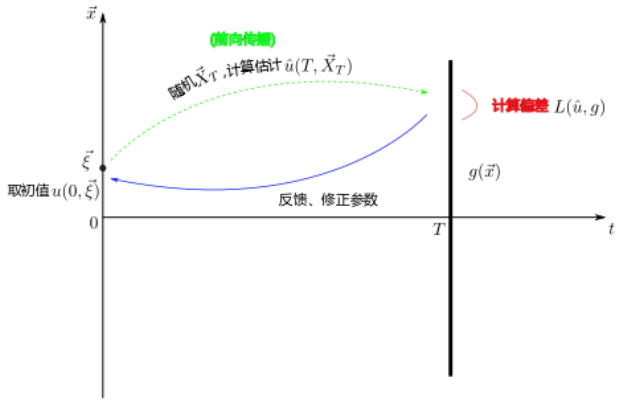

图 1: 求解思路

如上图所示，计算的思路是这样的：对于待求值 $u(0,\vec{\xi})$ 首先给一个“随机”初值，然后随机出空间上的变化量 $\vec{X_T}$，这样得到了 T 时刻的一个时空点 $(T,\vec{\xi}+\vec{X_T})$，利用微分方程和初值计算出这个点的估计值 $\hat{u}(T,\vec{\xi}+\vec{X_T})$，与终止条件在这个空间上的标准值 $g(\vec{\xi}+\vec{X_T})$ 比较计算出偏差，然后利用这个偏差修正初值与其他参数。整个过程分为前向传播和反馈修正两个过程。

1.3 随机过程

下面我们具体介绍是如何随机出空间上的变化量 $\vec{X_T}$ 的. 参考知乎专栏.

1.3.1 一维标准布朗运动 $B(t), t \geq 0$

定义:

1. $B(0) = 0$
2. 对所有 $0 < s < t, B(t) - B(s)$ 服从于 $N(0, t-s)$
3. 对不重叠的 $[s_i, t_i], B(t_i) - B(s_i)$ 相互独立

应该注意的是，在任意时刻 t，运动坐标的平均值 $\langle B(t) \rangle = 0$

1.3.2 带漂移的布朗运动

带漂移的布朗运动

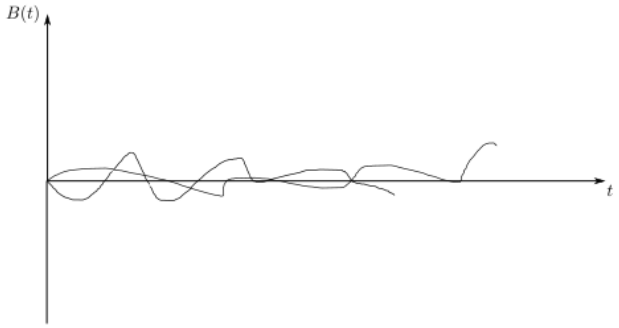

图 2: 一维标志布朗运动 $\{B(t), t \geq 0\}$

$$dX(t) = \mu dt + \sigma dB(t) \tag{2}$$

其中 σ 是尺度参数，μ 为常漂移系数.

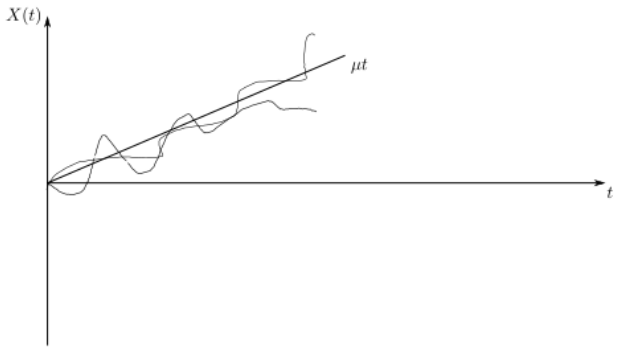

图 3: 带漂移的布朗运动 $\left\{X(t) \sim N(\mu t, \sigma^2 t)\right\}$

应该注意到这样的布朗运动有两个性质：

1. $X(t) \sim N(\mu t, \sigma^2 t)$
2. $\langle X(t) \rangle = \mu t$

1.3.3 文献中的随机运动及函数值之间的关联

文献中的随机运动由 [2] 式给出，这是一个 d 维随机过程

$$\vec{X}_t = \vec{\xi} + \int_0^t \vec{\mu}(s, \vec{X}_s)ds + \int_0^t \sigma(s, \vec{X}_s)d\vec{W}_s \tag{3}$$

而 [3] 式给出了待求函数值之间的关联

$$u(t, \vec{X}_t) - u(0, \vec{X}_0) = -\int_0^t f(s, \vec{X}_s, u(s, \vec{X}_s), \sigma^T(s, \vec{X}_s)\vec{\nabla}u(s, \vec{X}_s))ds + \int_0^t [\vec{\nabla}u(s, \vec{X}_s)]^t \sigma(s, \vec{X}_s)d\vec{W}_s \tag{4}$$

1.4 离散化

文献中 [4][5][6] 式给出了具体的离散化过程，

$$\vec{X}_{t_{n+1}} - \vec{X}_{t_n} \approx \vec{\mu}(t_n, \vec{X}_{t_n})\Delta t_n + \sigma(t_n, \vec{X}_{t_n})\Delta\vec{W}_n \tag{5}$$

$$u(t_{n+1}, \vec{X}_{t_{n+1}}) - u(t_n, \vec{X}_{t_n}) \approx -f(t_n, \vec{X}_{t_n}, u(t_n, \vec{X}_{t_n}), \sigma^T(t_n, \vec{X}_{t_n})\vec{\nabla}u(t_n, \vec{X}_{t_n}))\Delta t_n + [\vec{\nabla}u(t_n, \vec{X}_{t_n})]^T \sigma(t_n, \vec{X}_{t_n})\Delta\vec{W}_n \tag{6}$$

$$\Delta t_n = t_{n+1} - t_n, \Delta\vec{W}_n = \vec{W}_{t_{n+1}} - \vec{W}_{t_n} \tag{7}$$

1.5 例子

文献中有例子

$$\frac{\partial u(t, \vec{x})}{\partial t} = \Delta u(t, \vec{x}) + u(t, \vec{x}) - [u(t, \vec{x})]^3 \tag{8}$$

初始条件 $u(0, \vec{x}) = g(\vec{x}) = \dfrac{1}{2 + 0.4\|\vec{x}\|^2}$，求解 $u(t = 0.3, \vec{x} = \vec{0})$.

作变换 $t \to T - t$,

$$\frac{\partial u(t, \vec{x})}{\partial t} + \Delta u(t, \vec{x}) + u(t, \vec{x}) - [u(t, \vec{x})]^3 = 0 \tag{9}$$

终止条件 $u(T = 0.3, \vec{x}) = g(\vec{x}) = \dfrac{1}{2 + 0.4\|\vec{x}\|^2}$，求解 $u(t = 0, \vec{x} = \vec{0})$.

比照

$$\frac{\partial u}{\partial t}(t, \vec{x}) + \frac{1}{2}Tr(\sigma\sigma^T(t, \vec{x})(Hess_{\vec{x}}(t, \vec{x}))) + \vec{\nabla}u(t, \vec{x}) \cdot \vec{\mu}(t, \vec{x}) + f(t, \vec{x}, u(t, \vec{x}), \sigma^T(t, \vec{x})\vec{\nabla}u(t, \vec{x})) = 0 \tag{10}$$

取 $\sigma = \sigma^T = \begin{pmatrix} \sqrt{2} & & & \\ & \sqrt{2} & & \\ & & \ddots & \\ & & & \sqrt{2} \end{pmatrix} = \sqrt{2} \cdot I$

漂移 $\vec{\mu}(t, \vec{x}) = 0$，非线性 $f(t, \vec{x}, u(t, \vec{x}), \sigma^T(t, \vec{x})\vec{\nabla}u(t, \vec{x})) = u - u^3$.

对应的离散化为

$$\vec{X}_{t_{n+1}} - \vec{X}_{t_n} \approx \sqrt{2}\Delta\vec{W}_n \tag{11}$$

$$u(t_{n+1}, \vec{X}_{t_{n+1}}) - u(t_n, \vec{X}_{t_n}) \approx -(u(t_n, \vec{X}_{t_n}) - u(t_n, \vec{X}_{t_n})^3) \cdot \Delta t_n + \sqrt{2}[\vec{\nabla} u(t_n, \vec{X}_{t_n})]^T \cdot \Delta\vec{W}_n \tag{12}$$

$$\Delta t_n = t_{n+1} - t_n, \Delta\vec{W}_n = \vec{W}_{t_{n+1}} - \vec{W}_{t_n} \tag{13}$$

在这个例子中必须要注意的是 $\vec{\nabla} u(t_n, \vec{X}_{t_n})$ 这一项，这一项是真正使用神经网络求解的一项。

1.6 整个过程的信息流动方向

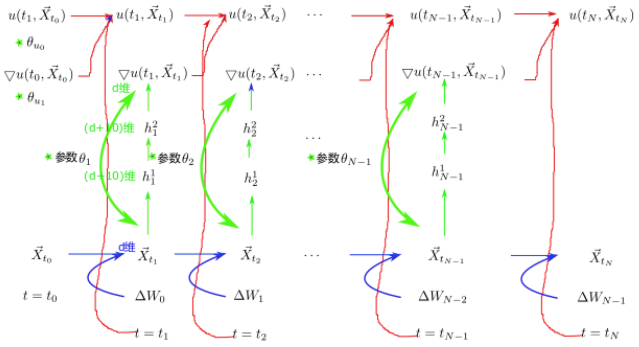

图 4: Flow of Information

2 主要结果的重复

主要重复了三个方程各自待求值的数值结果.

2.1 Nonlinear Black-Scholes Equation with Default Risk

2.1.1 建立模型

方程为

$$\frac{\partial u}{\partial t}(t, \vec{x}) + 0.02\vec{x} \cdot \vec{\nabla} u(t, \vec{x}) + 0.02 \sum_i |x_i|^2 \frac{\partial^2}{\partial x_i^2}(t, \vec{x}) - \frac{1}{3} Q(u(t, \vec{x})) \cdot u(t, \vec{x}) - 0.02u(t, \vec{x}) = 0 \tag{14}$$

其中函数

$$Q(x)=\begin{cases}0.2 & 50\le x\\ -0.009x+0.65 & 50<x<70\\ 0.02 & x\le 70\end{cases} \tag{15}$$

终止条件 $g(t=1,\vec{x})=\min\{x_1,x_2,\cdots,x_{100}\}$

取 $\sigma=\sigma^T=0.2\times\begin{pmatrix}x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_{100}\end{pmatrix}$

漂移 $\vec{\mu}(t,\vec{x})=0.02\vec{x}$，非线性 $f(t,\vec{x},u(t,\vec{x}),\sigma^T(t,\vec{x})\vec{\nabla}u(t,\vec{x}))=-\frac{1}{3}Q(u(t,\vec{x}))-0.02u(t,\vec{x})$.

对应的离散化为

$$\vec{X}_{t_{n+1}}-\vec{X}_{t_n}\approx 0.02\vec{x}\cdot\Delta t_n+0.2\begin{pmatrix}x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_{100}\end{pmatrix}\cdot\Delta\vec{W}_n \tag{16}$$

$$u(t_{n+1},\vec{X}_{t_{n+1}})-u(t_n,\vec{X}_{t_n})\approx\left[\frac{1}{3}Q(u(t_n,\vec{X}_{t_n}))\cdot u(t_n,\vec{X}_{t_n})+0.02u(t_n,\vec{X}_{t_n})\right]\cdot\Delta t_n+0.2[\vec{\nabla}u(t_n,\vec{X}_{t_n})]^T\begin{pmatrix}x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_{100}\end{pmatrix}\cdot\Delta\vec{W}_n \tag{17}$$

$$\Delta t_n=t_{n+1}-t_n,\Delta\vec{W}_n=\vec{W}_{t_{n+1}}-\vec{W}_{t_n} \tag{18}$$

2.1.2 具体求解过程

使用基于 tensorflow 的计算图进行求解，总的过程是定义计算图，开启会话进行初始化和求解.

引用包，定义变量

```
import tensorflow as tf
import numpy as np

dims=100
T=1.0
N=40
deltaT=T/N
std=0.0   #biaozhuicha

x=100*tf.ones([100],dtype=tf.float64)
```

```
u0=tf.Variable(58.5,dtype=tf.float64)
deltau0=tf.Variable(tf.random_normal([dims],stddev=std,dtype=tf.float64))
W1=tf.Variable(tf.random_normal([N-1,dims,dims+10],stddev=std,dtype=tf.float64))
B1=tf.Variable(tf.random_normal([N-1,dims+10],stddev=std,dtype=tf.float64))
W2=tf.Variable(tf.random_normal([N-1,dims+10,dims+10],stddev=std,dtype=tf.float64)
                                                    )
B2=tf.Variable(tf.random_normal([N-1,dims+10],stddev=std,dtype=tf.float64))
W3=tf.Variable(tf.random_normal([N-1,dims+10,dims],stddev=std,dtype=tf.float64))
B3=tf.Variable(tf.random_normal([N-1,dims],stddev=std,dtype=tf.float64))


batch_size=64
```

定义计算图. 计算图有 40 层，其中前两层为：

```
def g(x):
return tf.where(tf.less(x,50.0),0.2*tf.ones_like(x),tf.where(tf.greater(x,70.0),0.
                                                    02*tf.ones_like(x),0.2+(x-50.0)*(0.02-0.2
                                                    )/(70.0-50.0)))

#deltaW=tf.random_normal([batch_size,N,dims],stddev=std)    #mean=0,biaozhuicha=
                                                    stddev
deltaW=tf.random_normal([batch_size,N,dims],stddev=np.sqrt(deltaT),dtype=tf.
                                                    float64)
u1=u0+(g(u0)/3.0+0.02)*u0*deltaT+0.2*tf.reduce_sum(tf.multiply(tf.multiply(deltau0
                                                    ,x),deltaW[:,0,:]),axis=1)
x=x+0.02*deltaT*x+0.2*tf.multiply(x,deltaW[:,0,:])
xx1=tf.reshape(x,[batch_size,dims])
digits11=tf.matmul(xx1,W1[0])
###remember to add batch normalizatino
y11=tf.nn.relu(digits11)
digits21=tf.matmul(y11,W2[0])
y21=tf.nn.relu(digits21)
y31=tf.matmul(y21,W3[0])
deltau1=y31


u2=u1+(g(u1)/3.0+0.02)*u1*deltaT+0.2*tf.reduce_sum(tf.multiply(tf.multiply(tf.
                                                    reshape(deltau1,(64,100)),x),deltaW[:,1,:
                                                    ]),axis=1)
x=x+0.02*deltaT*x+0.2*tf.multiply(x,deltaW[:,1,:])
xx2=tf.reshape(x,[batch_size,-1,dims])
digits12=tf.matmul(xx2,W1[1])+B1[1]
###remember to add batch normalizatino
```

```
y12=tf.nn.relu(digits12)
digits22=tf.matmul(y12,W2[1])+B2[1]
y22=tf.nn.relu(digits22)
y32=tf.matmul(y22,W3[1])+B3[1]
deltau2=y32
```

然后按照第二层的模式一直重复到第 40 层

```
u38=u37+(g(u37)/3.0+0.02)*u37*deltaT+0.2*tf.reduce_sum(tf.multiply(tf.multiply(tf.
                                        reshape(deltau37,(64,100)),x),deltaW[:,37
                                        ,:]),axis=1)
x=x+0.02*deltaT*x+0.2*tf.multiply(x,deltaW[:,37,:])
xx38=tf.reshape(x,[batch_size,-1,dims])
digits138=tf.matmul(xx38,W1[37])+B1[37]
###remember to add batch normalizatino
y138=tf.nn.relu(digits138)
digits238=tf.matmul(y138,W2[37])+B2[37]
y238=tf.nn.relu(digits238)
y338=tf.matmul(y238,W3[37])+B3[37]
deltau38=y338


u39=u38+(g(u38)/3.0+0.02)*u38*deltaT+0.2*tf.reduce_sum(tf.multiply(tf.multiply(tf.
                                        reshape(deltau38,(64,100)),x),deltaW[:,38
                                        ,:]),axis=1)
x=x+0.02*deltaT*x+0.2*tf.multiply(x,deltaW[:,38,:])
xx39=tf.reshape(x,[batch_size,-1,dims])
digits139=tf.matmul(xx39,W1[38])+B1[38]
###remember to add batch normalizatino
y139=tf.nn.relu(digits139)
digits239=tf.matmul(y139,W2[38])+B2[38]
y239=tf.nn.relu(digits239)
y339=tf.matmul(y239,W3[38])+B3[38]
deltau39=y339


u40=u39+(g(u39)/3.0+0.02)*u39*deltaT+0.2*tf.reduce_sum(tf.multiply(tf.multiply(tf.
                                        reshape(deltau39,(64,100)),x),deltaW[:,39
                                        ,:]),axis=1)
x=x+0.02*deltaT*x+0.2*tf.multiply(x,deltaW[:,39,:])
```

创建会话并进行计算

```
y_=tf.reduce_min(x,reduction_indices=1)
cost=tf.losses.mean_squared_error(u40,y_)
```

```
optimizer=tf.train.AdamOptimizer(0.008).minimize(cost)

init=tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)

loss=np.array([])
ans=np.array([])
for i in range(2000):
#     print(sess.run(cost))
sess.run(optimizer)
loss=np.append(loss,sess.run(cost))
ans=np.append(ans,sess.run(u0))
print(i,sess.run(u0))
```

使用 matplotlib.pyplot 进行绘图

```
import matplotlib.pyplot as plt

plt.xlabel('iteration')
plt.ylabel('u(0,(100,100,...,100))')
plt.plot(ans)
plt.savefig('Black-Scholes equation ans.png')
plt.clf()
plt.xlabel('iteration')
plt.ylabel('~cost')
plt.plot(loss)
plt.savefig('Black-Scholes equation loss.png')
```

结果为

大致误差为

求得的函数值 $u(0,(100,100,...,100)) \approx 57.30$

2.2 HJB 方程

2.2.1 建立模型

方程为

$$\frac{\partial u}{\partial t}(t,\vec{x}) + \Delta u(t,\vec{x}) - \|\vec{\nabla} u(t,\vec{x})\|^2 = 0 \tag{19}$$

终止条件 $g(\vec{x}) = \ln(\frac{1+\|\vec{x}\|^2}{2})$

取 $\sigma = \sigma^T = \sqrt{2} \cdot I$

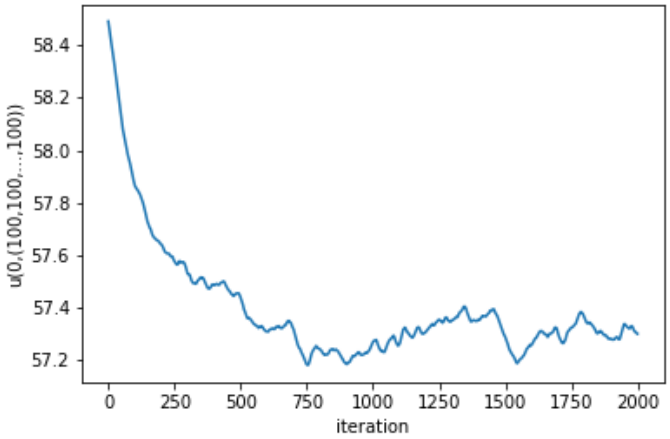

图 **5:** Black-Scholes equation ans

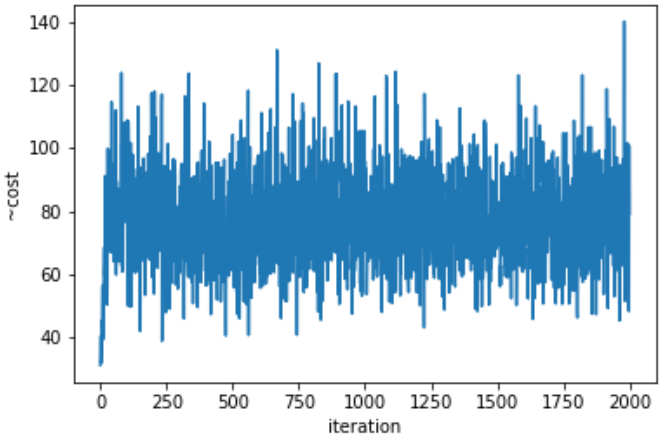

图 **6:** Black-Scholes equation loss

漂移 $\vec{\mu}(t,\vec{x})=0$，非线性 $f(t,\vec{x},u(t,\vec{x}),\sigma^T(t,\vec{x})\vec{\nabla}u(t,\vec{x}))=-\|\vec{\nabla}u(t,\vec{x})\|^2$.

对应的离散化为

$$\vec{X}_{t_{n+1}}-\vec{X}_{t_n}\approx\sqrt{2}\cdot\Delta\vec{W}_n \tag{20}$$

$$u(t_{n+1},\vec{X}_{t_{n+1}})-u(t_n,\vec{X}_{t_n})\approx\|\vec{\nabla}u(t_n,\vec{X}_{t_n})\|^2\cdot\Delta t_n+\sqrt{2}\vec{\nabla}u(t_n,\vec{X}_{t_n})\cdot\Delta\vec{W}_n \tag{21}$$

$$\Delta t_n=t_{n+1}-t_n,\Delta\vec{W}_n=\vec{W}_{t_{n+1}}-\vec{W}_{t_n} \tag{22}$$

求解 $u(t=0,\vec{x}=\vec{0})$.

2.2.2 具体求解过程

引用包，定义变量

```
import tensorflow as tf
import numpy as np

dims=100
T=1.0
N=20
deltaT=T/N
std=0.0  #biaozhuicha

x=tf.zeros([dims],dtype=tf.float64)
u0=tf.Variable(1.0,dtype=tf.float64)
deltau0=tf.Variable(tf.random_normal([dims],stddev=std,dtype=tf.float64))
W1=tf.Variable(tf.random_normal([N-1,dims,dims+10],stddev=std,dtype=tf.float64))
W2=tf.Variable(tf.random_normal([N-1,dims+10,dims+10],stddev=std,dtype=tf.float64)
                                                    )
W3=tf.Variable(tf.random_normal([N-1,dims+10,dims],stddev=std,dtype=tf.float64))
B=tf.Variable(tf.random_normal([N-1,dims],stddev=std,dtype=tf.float64))



batch_size=64
```

定义计算图. 前两层为

```
#deltaW=tf.random_normal([batch_size,N,dims],stddev=std)   #mean=0,biaozhuicha=
                                                    stddev
deltaW=tf.random_normal([batch_size,N,dims],stddev=np.sqrt(deltaT),dtype=tf.
                                                    float64)
u1=u0+tf.reduce_sum(tf.square(deltau0))*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.
                                                    multiply(deltau0,deltaW[:,0,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,0,:]
```

```
xx1=tf.reshape(x,[batch_size,dims])
digits11=tf.matmul(xx1,W1[0])
bat11=tf.layers.batch_normalization(digits11,axis=-1,training=True)
y11=tf.nn.relu(bat11)
digits21=tf.matmul(y11,W2[0])
bat21=tf.layers.batch_normalization(digits21,axis=-1,training=True)
y21=tf.nn.relu(bat21)
y31=tf.matmul(y21,W3[0])
deltau1=y31+B[0]


u2=u1+tf.reduce_sum(tf.square(deltau1),axis=1)*deltaT+np.sqrt(2.0)*tf.reduce_sum(
                                        tf.multiply(deltau1,deltaW[:,1,:]),axis=1
                                        )
x=x+np.sqrt(2.0)*deltaW[:,1,:]
xx2=tf.reshape(x,[batch_size,dims])
digits12=tf.matmul(xx2,W1[1])
bat12=tf.layers.batch_normalization(digits12,axis=-1,training=True)
y12=tf.nn.relu(digits12)
digits22=tf.matmul(y12,W2[1])
bat22=tf.layers.batch_normalization(digits22,axis=-1,training=True)
y22=tf.nn.relu(digits22)
y32=tf.matmul(y22,W3[1])+B[1]
deltau2=y32
```

一直重复到第 20 层

```
u19=u18+tf.reduce_sum(tf.square(deltau18),axis=1)*deltaT+np.sqrt(2.0)*tf.
                                        reduce_sum(tf.multiply(deltau18,deltaW[:,
                                        18,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,18,:]
xx19=tf.reshape(x,[batch_size,dims])
digits119=tf.matmul(xx19,W1[18])
bat119=tf.layers.batch_normalization(digits119,axis=-1,training=True)
y119=tf.nn.relu(digits119)
digits219=tf.matmul(y119,W2[18])
bat219=tf.layers.batch_normalization(digits219,axis=-1,training=True)
y219=tf.nn.relu(digits219)
y319=tf.matmul(y219,W3[18])+B[18]
deltau19=y319


u20=u19+tf.reduce_sum(tf.square(deltau19),axis=1)*deltaT+np.sqrt(2.0)*tf.
                                        reduce_sum(tf.multiply(deltau19,deltaW[:,
```

```
                                                    19,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,19,:]
```

定义损失函数及递归算法

```
y_=tf.log((1.0+tf.reduce_sum(tf.square(x),axis=1))/2.0)
cost=tf.losses.mean_squared_error(u20,y_)
optimizer=tf.train.AdamOptimizer(0.01).minimize(cost)
```

创建会话并开始计算

```
init=tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)

loss=np.array([])
ans=np.array([])
for i in range(2000):
#     print(sess.run(cost))
sess.run(optimizer)
loss=np.append(loss,sess.run(cost))
ans=np.append(ans,sess.run(u0))
print(i,sess.run(u0))
```

结果及大致误差为

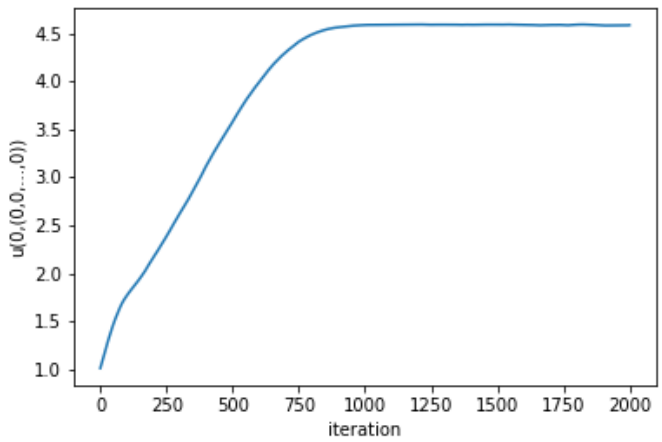

图 7: HJB equation ans

最终得到的函数值 $u(0,\vec{0}) \approx 4.59$

2.3 Allen-Cahn Equation

第一部分中的例子即为文献中的这个方程，下面直接介绍求解的具体步骤。

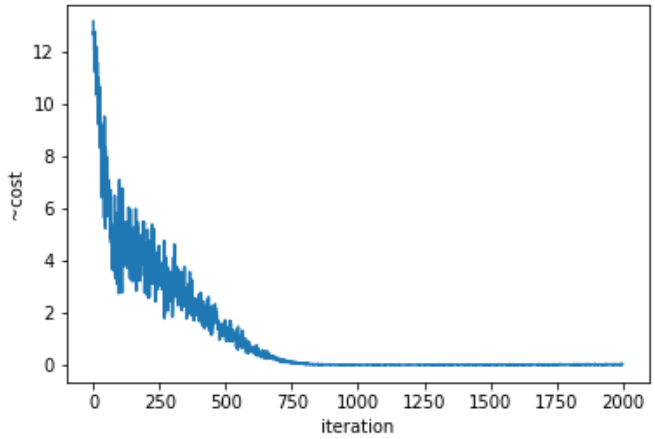

图 8: HJB equation loss

引用包，定义变量

```
import tensorflow as tf
import numpy as np

dims=100
T=0.3
N=20
deltaT=T/N
std=0.0  #biaozhuicha

x=tf.zeros([dims],dtype=tf.float64)
u0=tf.Variable(1.0,dtype=tf.float64)
deltau0=tf.Variable(tf.random_normal([dims],stddev=std,dtype=tf.float64))
W1=tf.Variable(tf.random_normal([N-1,dims,dims+10],stddev=std,dtype=tf.float64))
W2=tf.Variable(tf.random_normal([N-1,dims+10,dims+10],stddev=std,dtype=tf.float64)
                                                    )
W3=tf.Variable(tf.random_normal([N-1,dims+10,dims],stddev=std,dtype=tf.float64))
B=tf.Variable(tf.random_normal([N-1,dims],stddev=std,dtype=tf.float64))


batch_size=64
```

定义计算图. 前两层为

```
deltaW=tf.random_normal([batch_size,N,dims],stddev=np.sqrt(0.3/20),dtype=tf.
                                                    float64)
u1=u0-(u0-u0**3.0)*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.multiply(deltau0,deltaW[:,
                                                    0,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,0,:]
```

```
xx1=tf.reshape(x,[batch_size,dims])
digits11=tf.matmul(xx1,W1[0])
bat11=tf.layers.batch_normalization(digits11,axis=-1,training=True)
y11=tf.nn.relu(bat11)
digits21=tf.matmul(y11,W2[0])
bat21=tf.layers.batch_normalization(digits21,axis=-1,training=True)
y21=tf.nn.relu(bat21)
y31=tf.matmul(y21,W3[0])+B[0]
deltau1=y31



u2=u1-(u1-u1**3.0)*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.multiply(deltau1,deltaW[:,
                                        1,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,1,:]
xx2=tf.reshape(x,[batch_size,dims])
digits12=tf.matmul(xx2,W1[1])
bat12=tf.layers.batch_normalization(digits12,axis=-1,training=True)
y12=tf.nn.relu(digits12)
digits22=tf.matmul(y12,W2[1])
bat22=tf.layers.batch_normalization(digits22,axis=-1,training=True)
y22=tf.nn.relu(digits22)
y32=tf.matmul(y22,W3[1])+B[1]
deltau2=y32
```

一直重复到第 20 层

```
u18=u17-(u17-u17**3.0)*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.multiply(deltau17,
                                        deltaW[:,17,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,17,:]
xx18=tf.reshape(x,[batch_size,dims])
digits118=tf.matmul(xx18,W1[17])
bat118=tf.layers.batch_normalization(digits118,axis=-1,training=True)
y118=tf.nn.relu(digits118)
digits218=tf.matmul(y118,W2[17])
bat218=tf.layers.batch_normalization(digits218,axis=-1,training=True)
y218=tf.nn.relu(digits218)
y318=tf.matmul(y218,W3[17])+B[17]
deltau18=y318


u19=u18-(u18-u18**3.0)*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.multiply(deltau18,
                                        deltaW[:,18,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,18,:]
```

```
xx19=tf.reshape(x,[batch_size,dims])
digits119=tf.matmul(xx19,W1[18])
bat119=tf.layers.batch_normalization(digits119,axis=-1,training=True)
y119=tf.nn.relu(digits119)
digits219=tf.matmul(y119,W2[18])
bat219=tf.layers.batch_normalization(digits219,axis=-1,training=True)
y219=tf.nn.relu(digits219)
y319=tf.matmul(y219,W3[18])+B[18]
deltau19=y319


u20=u19-(u19-u19**3.0)*deltaT+np.sqrt(2.0)*tf.reduce_sum(tf.multiply(deltau19,
                                                deltaW[:,19,:]),axis=1)
x=x+np.sqrt(2.0)*deltaW[:,19,:]
```

定义会话并进行计算

```
y_=1.0/(2.0+0.4*tf.reduce_sum(tf.square(x),axis=1))
cost=tf.losses.mean_squared_error(u20,y_)
optimizer=tf.train.AdamOptimizer(0.005).minimize(cost)

init=tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)

loss=np.array([])
ans=np.array([])
for i in range(2000):
sess.run(optimizer)
ans=np.append(ans,sess.run(u0))
loss=np.append(loss,sess.run(cost))
print(i,sess.run(u0))
```

结果及大致误差为

相应的误差大致为

求得结果为 $u(0.3,\vec{0}) \approx 0.0529$

代码可以在Github这里找到.

3 存在的问题

文章重复了文献中的最重要的结果. 存在的问题还有:

1. 文献中对同一个问题进行了 5 次独立的随机计算. 本文中存在的问题是由于层数比较多 (约 20 或 40 层)，使得最初的小量在经过层级放大之后会变得无穷大，无法进行计算. 只能采用将所

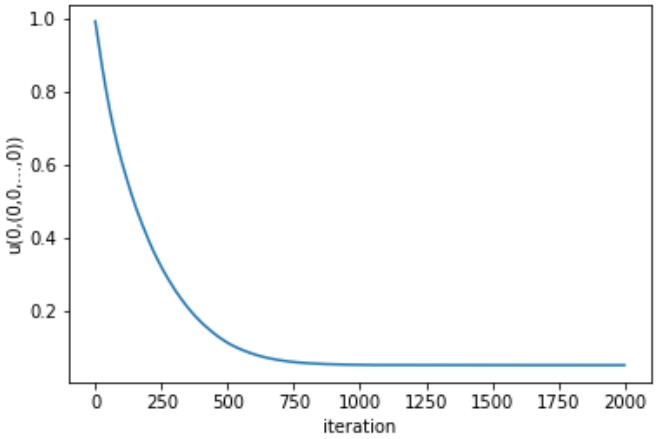

图 **9:** Allen-Cahn equation ans

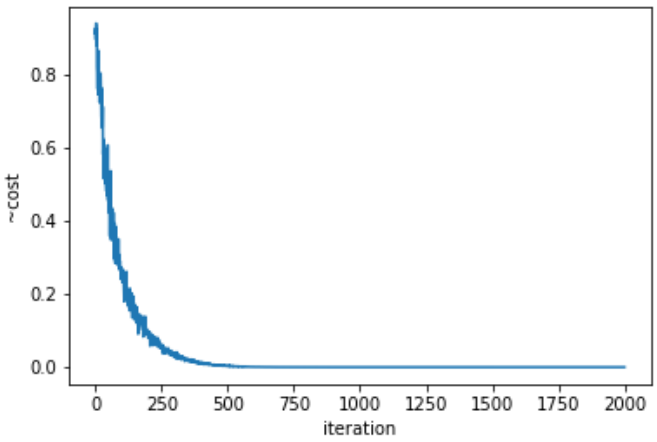

图 **10:** Allen-Cahn equation loss

有参数值最初都设置为零的方法 (文献是使用正态分布进行随机)，这样带来的问题是只能进行一次有效计算.

2. 文献中使用了一个叫做 batch_normalization 的神经网络中的归一化技术. 本文中存在的问题为由于第一点只能进行一次计算,似乎没有进行 batch_normalization 的意义,同时对 batch_normalization 的原理和具体使用细节也有待考量.
3. 没有重复进一步的几个时间的计算等.

4 与作者代码的比较

在这里Github可以找到作者的代码. 作者的代码写得要漂亮老道地多.

主要是四个文件.

1. 在 main.py 中类似定义了一个用户交互界面，只需要输入求解的方程名称即可开始进行计算. 其他文件中的内容都已 Class 的形式存在.
2. 在 equation.py 中写求解的方程 (如 AllenCahn 方程、HJB 方程) 的信息 (如非线性函数、终止条件).
3. 在 solver.py 中定义网络结构、训练步骤等.
4. 在 config.py 中定义配置信息 (如维度、迭代次数、时间间隔).

5 灵魂

目标都是求出待求函数的估计. 方法都是前馈给出估计值和标准差之间的偏差，然后由偏差反馈修改参数. 玩的花样是使用不同的网络结构.

5.1 Deep Galerkin Method

在时空域上进行采样. 误差由待求函数、边界条件和初始条件三部分构成. 网络包含多个 DGM 层，每层中有 Z、G、R、H 几个值.(如何进行初始化?)

5.2 PNAS

(主要) 关注单个点的求解. 误差由终止条件构成. 在时空域上进行随机.神经网络的作用仅仅是由空间坐标求解函数在该点的微分.