TensorFlow 优化实践

写在前面的话

在前面一章中说到了TensorFlow的基础用法，这一章作为一个进阶来聊聊神经网络的具体的结构和参数问题，包括：

1. 前馈神经网络
2. 循环神经网络
3. 神经网络参数

大概就这三个中心内容。当然每个部分展开又是很多东西，但是好在这里只是聊聊工具的使用。所以大概还是一个章节的内容。

前馈神经网络

前馈神经网络包括全链接网络与卷积神经网络，这两者其实并没有大的区别，一般情况下可以将卷积神经网络看做是一个特例。

全链接网络与卷积网络

霍金说过科普书里加一个公式就会少一半读者，所以这里用公式来直观的说明二者的共同点：

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}_{6\times6}$$

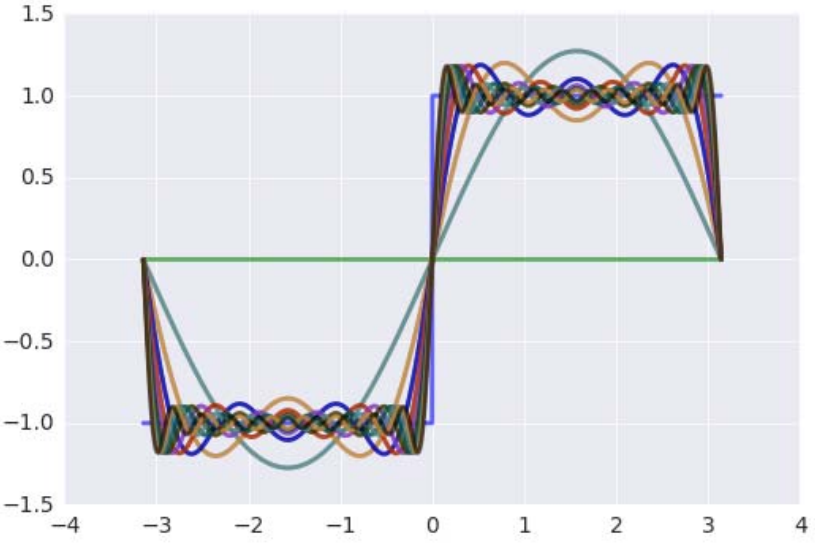
前面说到全链接网络是一个矩阵相乘的过程：

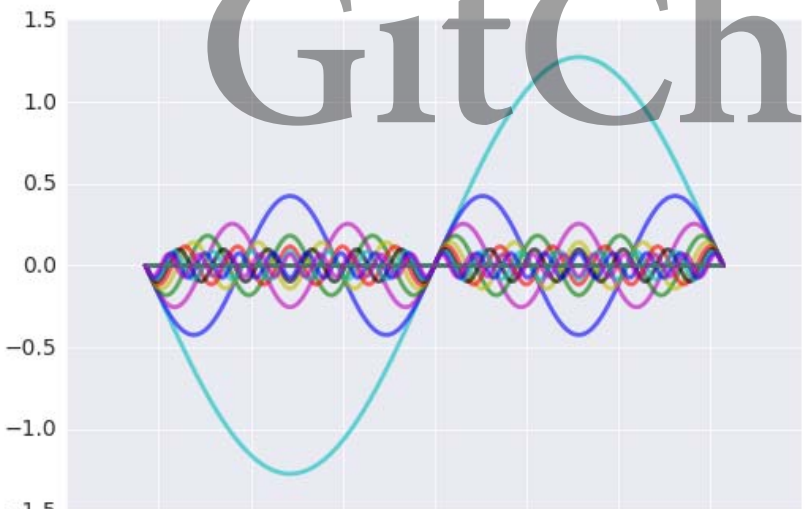
$$y = ActiveFunction(x \cdot A)$$

得到了有效的减少，另外由于序列数据或图像数据在计算过程中特征的相似性，使得各个部分的特征具有重复性，这样在计算中就可以利用同一套权值：

$$A = \left[\frac{1}{3} \quad \frac{1}{3} \quad \frac{1}{3}\right]_{1\times 3}$$

这就是所谓的**权值共享**，通过这个过程，就可以使得全连接层中的参数量极大的减少。这个就是全连接网络和卷积网络的共同点。提一下，上面那个卷积核心可以计算的特征是“滑动平均”。
接下来看下函数的傅里叶展开：





```
y = tf.nn.conv1d(x,kernel,stride=1,padding="SAME")
sess=tf.Session()
sess.run(tf.global_variables_initializer())
print("Out:",sess.run(y))
print("Shape:",np.shape(sess.run(y)))
```

这是一个长为1000的一维向量，其特征只有一个最前面的1代表输入数据个数为1，卷积核心维度为[3,1,2]，3代表卷积核心大小，也就是前面所说的能覆盖多少数据点，1代表输入数据特征数量，2是输出数据特征数量，可以看到，对于第0个特征其卷积核心的权值为$\frac{1}{3}$也就是求取均值，第二个卷积核心权值为1，也就是对三个数据进行相加，可以看下最终结果：

```
Out: [[[ 0.66666669  2.        ]
  [ 1.          3.        ]
  [ 1.          3.        ]
  ...,
  [ 1.          3.        ]
  [ 1.          3.        ]
  [ 0.66666669  2.        ]]]
Shape: (1, 1000, 2)
```

可以看到，输出数据特征并没有什么违反直觉的地方，输出为两个特征，第一个特征为均值，第二个为相加，由于padding边界设置为“SAME”，也就是从第一个点开始作为输出点，这就使得超出部分的权值、数据赋为0。

当然在神经网络之中前面Variable是需要自适应的变化调整以提取特定数据的特征。

tensorflow构建前馈神经网络

用tensorflow构建前馈神经网络总共分为三种方式：

第一种：直接构建


```
import tensorflow as tf
#卷积层
def conv1d_layer(input_tensor, kernel_size, feature=2, active_function="relu",
name='conv1d'):
    activ={"relu":tf.nn.relu,"sigmoid":tf.nn.sigmoid,"tanh":tf.nn.tanh}
    with tf.variable_scope(name):
        shape = input_tensor.get_shape().as_list()
        kernel = tf.get_variable('kernel',
                                 (kernel_size, shape[-1], feature),
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0))
```

```
                                [out_dim],
                                dtype=tf.float32,
                                initializer=tf.constant_initializer(0))
            out = tf.matmul(input_tensor,W) + b
        return activ[active_function](out)
```

上面是用tensorflow函数构建的全链接网络与卷积神经网络。函数调用过程可以使用：

```
net=full_layer(xx, featrue=2, active_function="relu", name="full_connect_1")
net=conv1d_layer(net,3, feature=2, active_function="relu", name="conv1d_1")
```

这样就构成了单一层神经网络的构建。

对于多层神经网络而言可以将输出数据进行进行卷积和全链接操作：

```
net = conv1d_layer(net, 3, feature=2, active_function="relu", name="conv1d_1")
net = conv1d_layer(net, 3, feature=2, active_function="relu", name="conv1d_2")
net=conv1d_layer(net, 3, feature=2, active_function="relu", name="conv1d_3")
```

当然每层的name要不同，否则会报错。

第二种：通过contrib高层次API

前面用一些基本的API构建过程需要自行的去写一些函数，但是这个函数是可以通过contrib里面的函数去进行简单的构建的：


```
import tensorflow.contrib.slim as slim
net = slim.conv2d(net, 3, 1, scope='conv1d_1')
net = slim.flatten(net)
net = slim.flatten(net)
net = slim.fully_connected(net, 2,
                           activation_fn=tf.nn.relu,
                           scope='outes',
                           reuse=False)
```

可以看到通过slim高层次api实际上可以简化整个网络的构建过程。这里有个reuse参数，是复用参数。

通过keras构建神经网络

keras其实是一个构建神经网络比较方便的函数库，其底层计算可以放到tensorflow之中：

而循环神经网络与上一步的输入相关，其实只是乘以了一个上一步的矩阵：

$$h_t = ActiveFunction(h_{t-1} \cdot A_1 + x_t \cdot A_2)$$
$$y_t = h_t \cdot A_3$$

这就是一个最简单的循环神经网络，在此之上可以构建多层的神经网络，定义y_t与x_t之间的关系为：

$$y_t = func_t(x_t)$$
$$z_t = func_t(y_t)$$

这是构建的多层的循环神经网络。其他的循环神经网络结构是在其上进行的修改。只是参数更多，比如LSTM。LSTM输入中加入了一个C矩阵用于保存记忆。

$$y_t, c_t = LSTM(x_t, c_{t-1})$$

tensorflow构建循环神经网络

上面说到一个循环神经网络的层可以通过如下的函数构建：

```
cell = tf.contrib.rnn.BasicLSTMCell(
           hidden_size, forget_bias=0.0, state_is_tuple=True,
           reuse=not is_training)
```

多层卷积神经网络构建：


```
cell = tf.contrib.rnn.MultiRNNCell(
        [cell for _ in range(num_layers)], state_is_tuple=True)
```

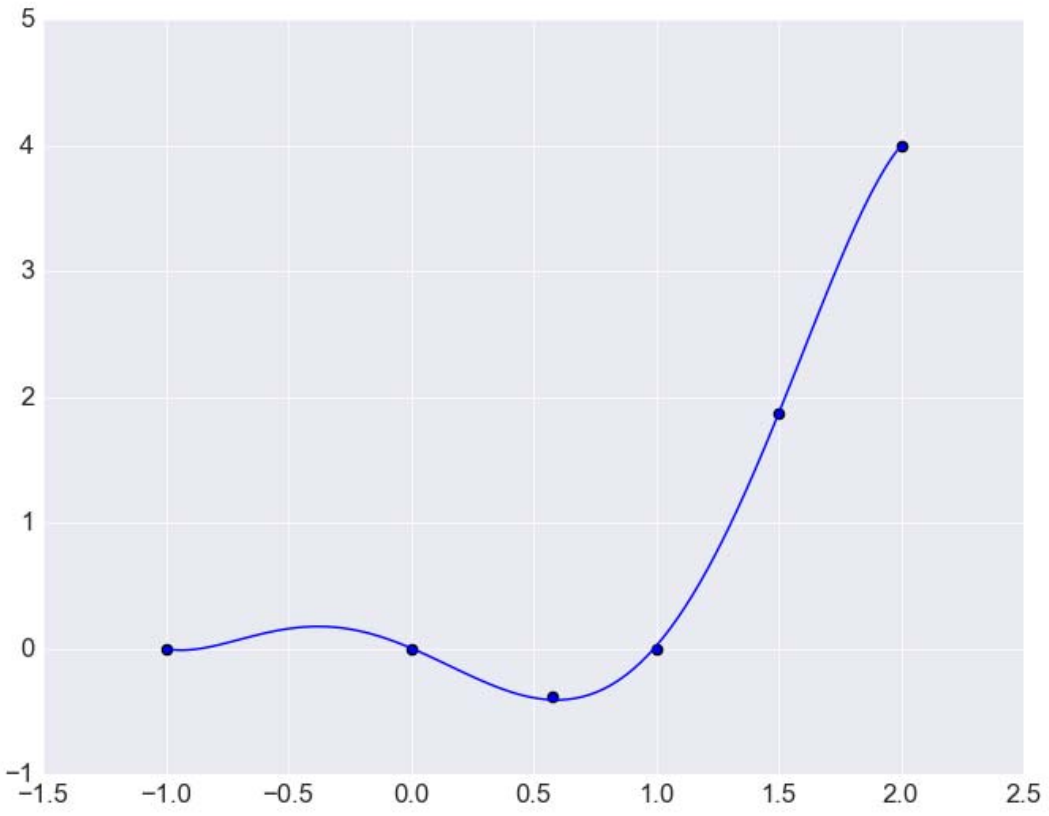
前面说到循环神经网络的数据输入是一步一步输入的，因此这里在进行数据输入的过程中也需要一步一步的进行数据的输入：

```
for time_step in range(num_steps):
    if time_step > 0: tf.get_variable_scope().reuse_variables()
    (cell_output, state) = cell(inputs[:, time_step, :], state)
```

这里有两个地方需要解释，第一个是reuse的问题，因为在第0步之后的变量与前面的变量都是复用的，因此这里需要将其改变为复用模式。第二个是state，这个state是LSTM神经网络结构所特有的，其目的在于保存“记忆”，而记忆

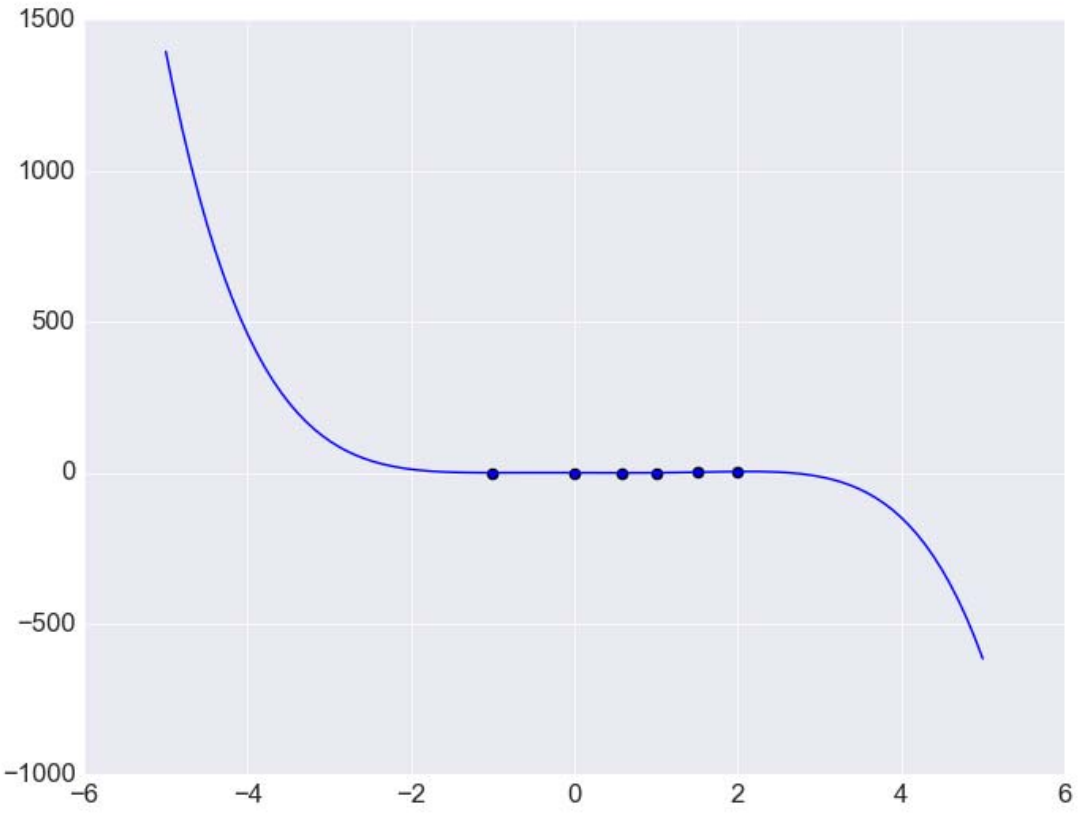
```
import tensorflow as tf
import numpy as
#定义函数展开阶数
N = 6
x = tf.placeholder(dtype=tf.float32,shape=[1,None])
y = tf.placeholder(dtype=tf.float32,shape=[1,None])
comp = []
#函数展开项
for itr in range(N):
    comp.append(tf.pow(x, itr))
x_v = tf.concat(comp,axis=0)
#定义展开系数
A = tf.Variable(tf.zeros([1, N]))
y_new = tf.matmul(A, x_v)
#定义loss函数
loss = tf.reduce_sum(tf.square(y-y_new))
#用梯度迭代法求解
train_step = tf.train.GradientDescentOptimizer(0.0005).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
#迭代9000次
for itr in range(90000):
    sess.run(train_step,
             feed_dict={x:np.array([[-1, 0, 0.5773502691896258, 1, 1.5, 2]]),
                        y:np.array([[0, 0, -0.3849, 0, 1.875, 4]])})
print(sess.run(A.value()))
#图形绘制
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.style.use('seaborn-darkgrid')
lin = np.array([np.linspace(-1, 2, 100)])
ly =sess.run(y_new, feed_dict={x: lin})
plt.plot(lin[0], ly[0])
plt.scatter([-1, 0, 0.5773502691896258, 1, 1.5, 2],
            [0, 0, -0.3849, 0, 1.875, 4])
plt.show()
```


来看下输出图形：



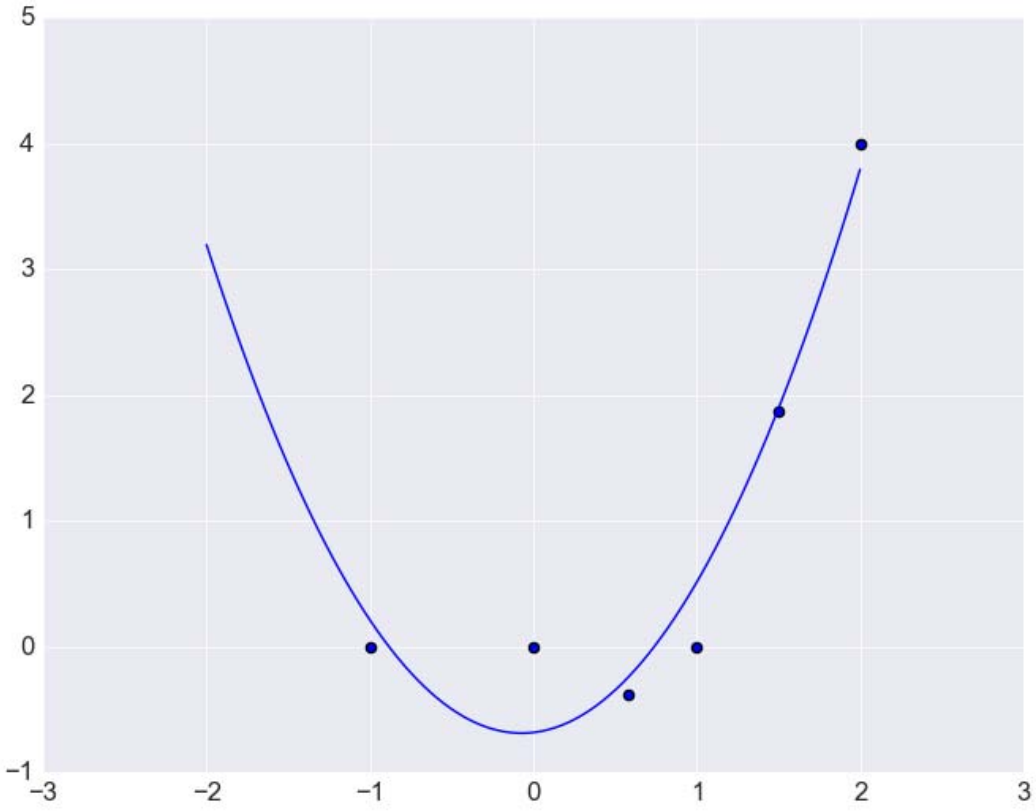
可以看到曲线完美的通过了每一个点，但是这种情况可能并非是一个“好”的结果，因为其对于训练数据拟合过于好了，对于训练集以外的数据可能预测效果并不理想。将函数取值范围从-5到5再来绘制图形：


```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.style.use('seaborn-darkgrid')
lin = np.array([np.linspace(-5, 5, 100)])
ly =sess.run(y_new, feed_dict={x: lin})
plt.plot(lin[0], ly[0])
plt.scatter([-1, 0, 0.5773502691896258, 1, 1.5, 2],
            [0, 0, -0.3849, 0, 1.875, 4])
plt.show()
```

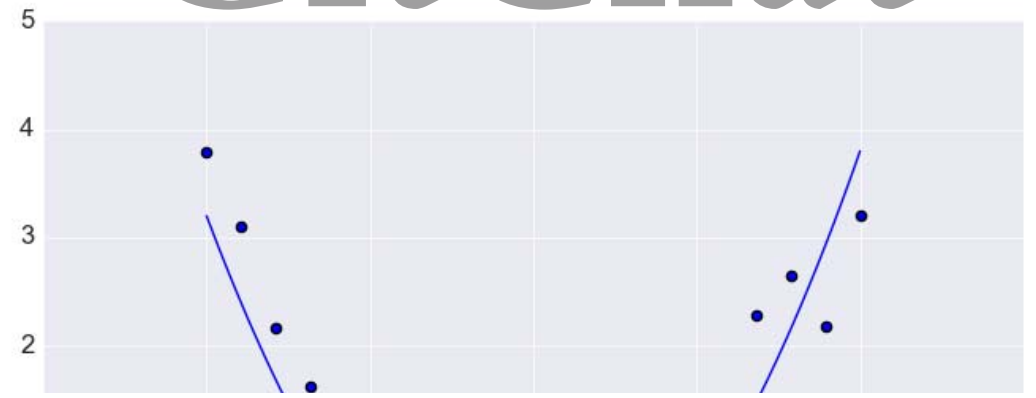


在训练数据之外曲线变形比较大，这会影响对于曲线的拟合效果。另一方面来说也就是对于数据的预测效果不理想。在这里如果减少参数数量：

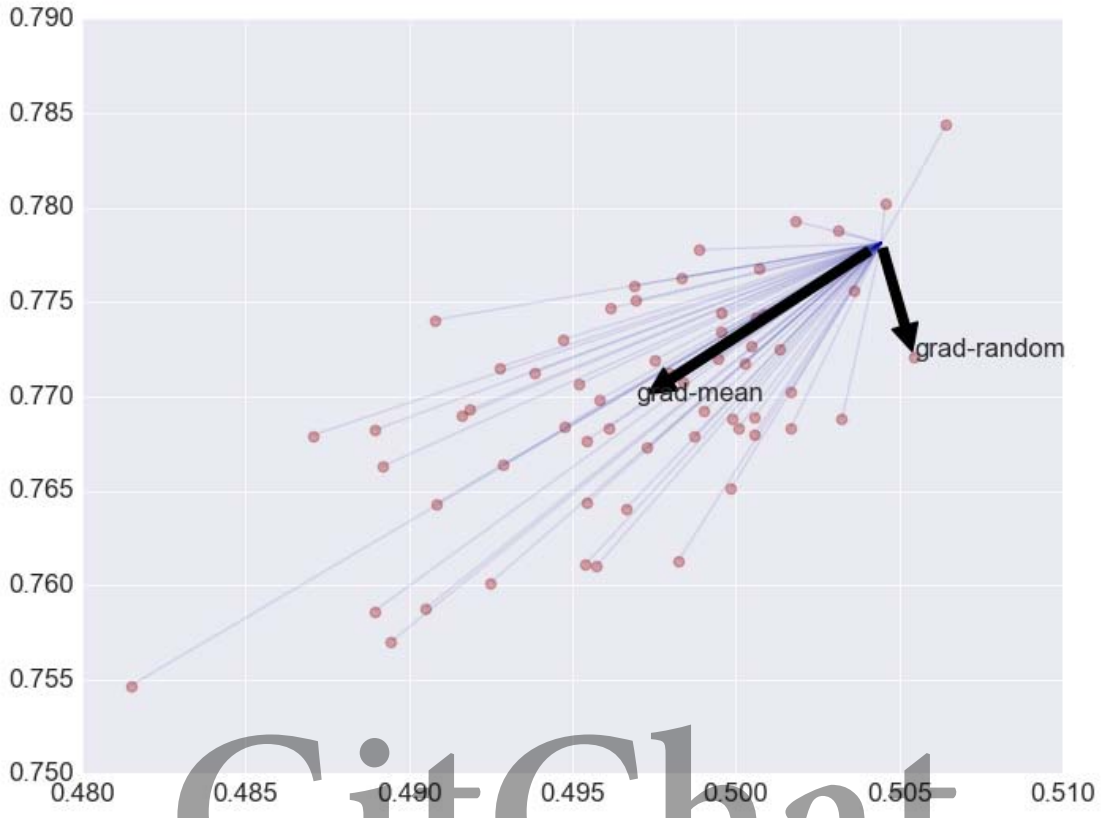

```
N=3
```



此时可能并未通过每一个点，但是所用的自由参数个数只有三个，但是对于可能的数据分布形式：

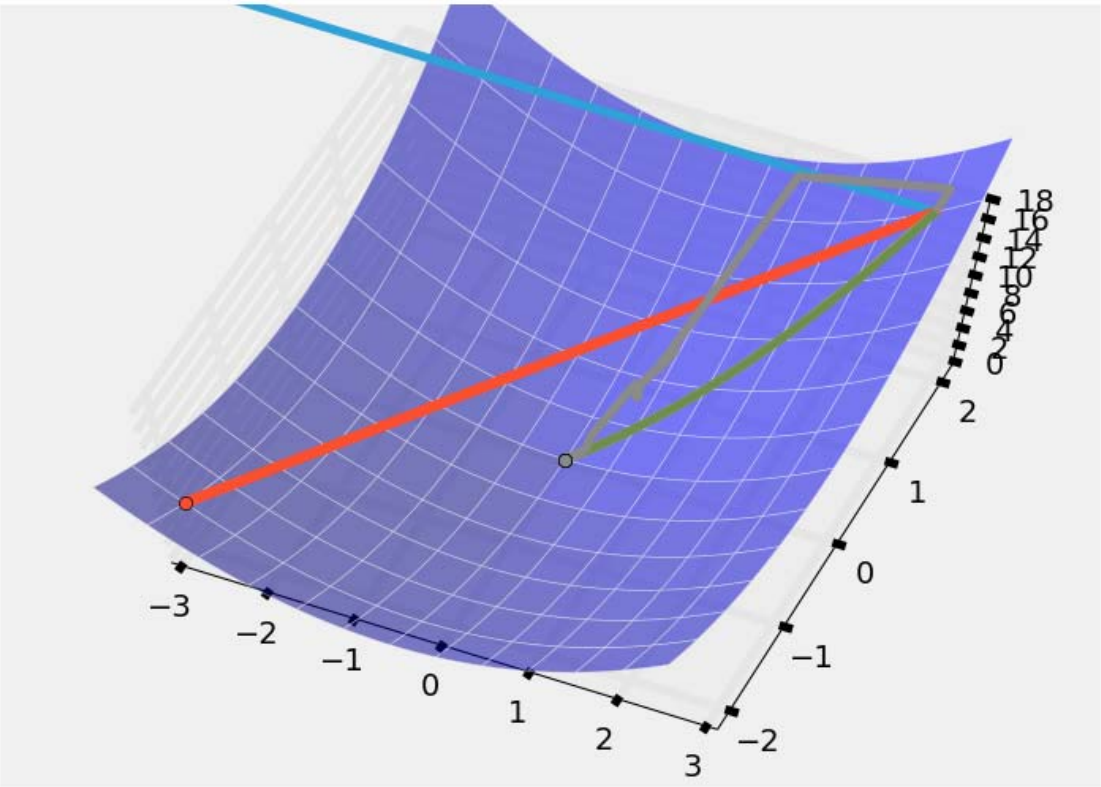


这个是一次输入多少数据用于训练的问题，下面的梯度方向是基于二维曲面拟合的LMS方法所言的，可以看到：



随机梯度带有一定的随机性，虽然按照概率角度来说是可以收敛的，但是显然通过60个数据进行梯度的估计所带来的方向预测更加准确，因此BATCHSIZE存在的一个意义就是进行更好的梯度方向的预测，当然如果用整个训练集来估计梯度方向的话是更好的。但是这里面临一个效率和内存的问题。这里用17层Inception网络来做个表格，实验机器就是E3非GPU版本：

BATCHSIZE	最大内存	单次迭代时间
50	1534	8.7S
100	2096	19.2S
200	3392	37.3s

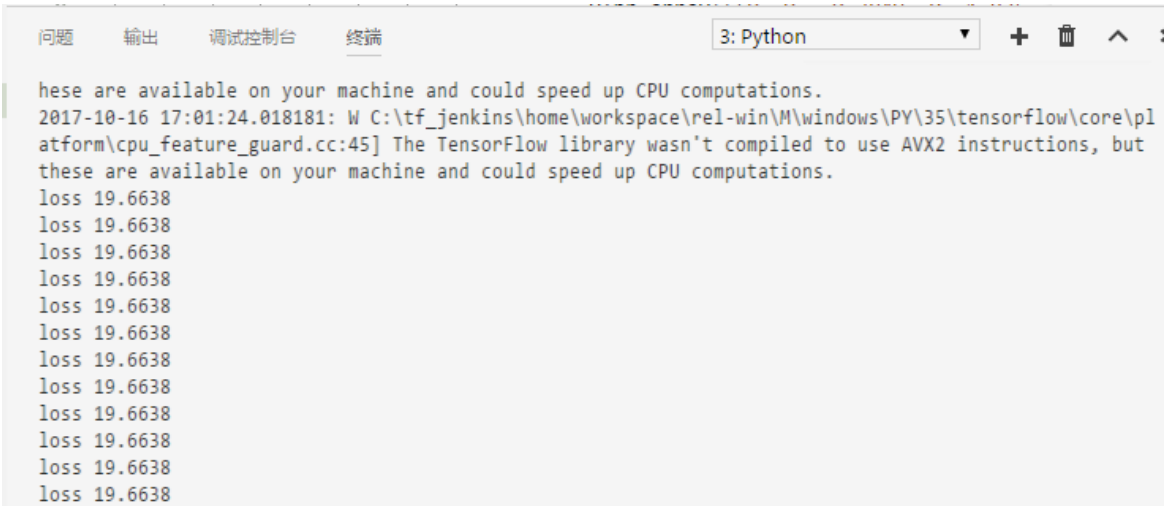


那个蓝线，在计算过程中不仅没有逐渐减少反而越来越大，想理解发散过程比较简单：

```
train_step = tf.train.GradientDescentOptimizer(500).minimize(loss)
```

把曲线拟合的梯度参数选为500，我这里选择的是0.5来观察loss函数：

```
loss 16714.6
loss 1.70027e+07
loss 1.73021e+10
loss 1.76068e+13
loss 1.79169e+16
loss 1.82325e+19
loss 1.85536e+22
loss 1.88803e+25
loss 1.92129e+28
loss 1.95512e+31
loss 1.98955e+34
```



收敛过程变得极为缓慢，或者说基本没有变化。

理智的做法梯度步长随着迭代不断的减少。

BATCHNORM层

前面说过BP算法中一个过程就是乘以矩阵：有兴趣参考文章

但随着层数增多所乘的矩阵也在不断的增多，那么就会遇到梯度消失的问题，就像鸡汤说的每天忘记百分之一，一百天后就剩下37%了，这是一个指数的问题，BP算法也是如此，因此引入了BATCHNORM层，以解决上述问题。
tensorflow中有相应的代码：

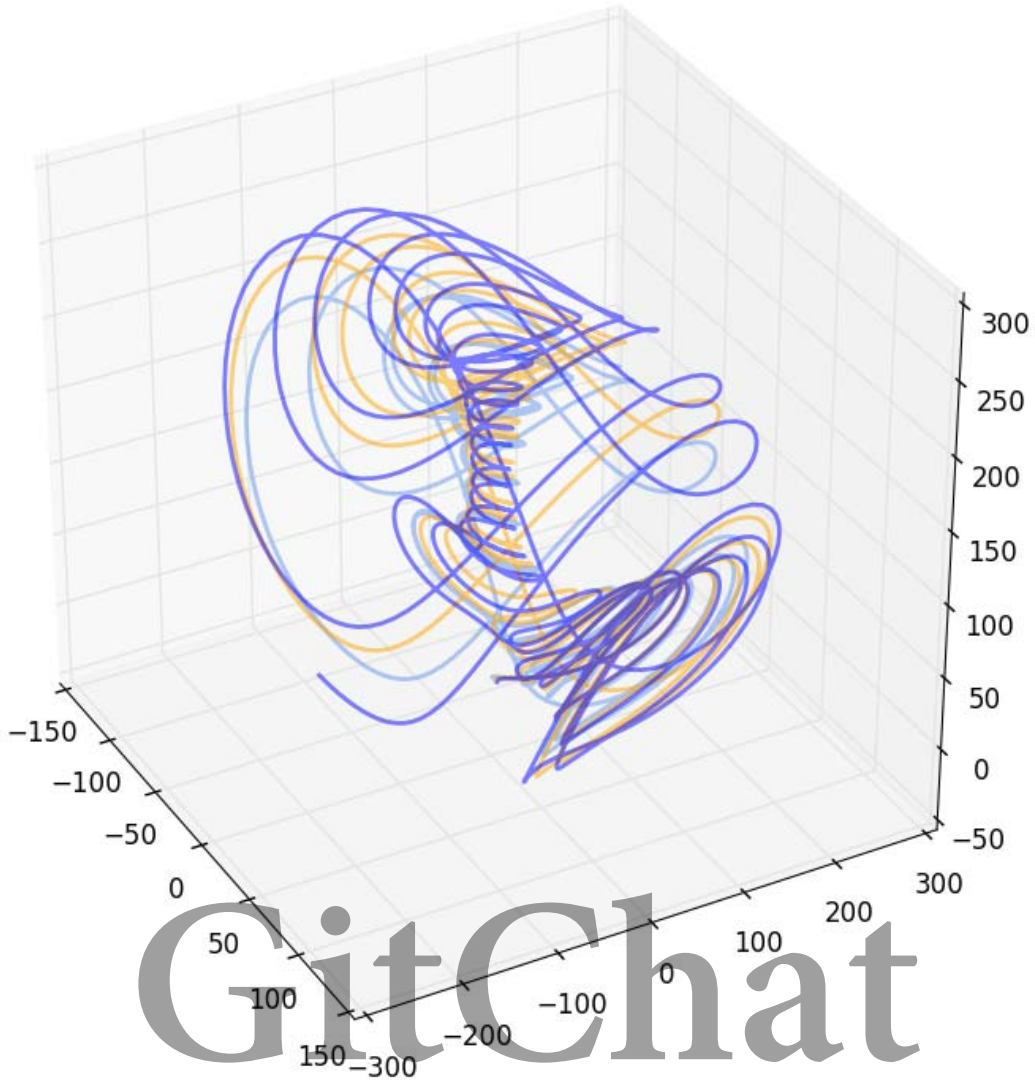
```
tf.contrib.layers.batch_norm
```


形象化

一般在讨论纯理论问题的时候很多人都有个**很不好**的习惯就是找一个**物理映射**，这种物理映射可以帮助理解问题，但是可能会在之后的学习中并不如直接的**数学理解**更有帮助。
那么再来看下整个的循环神经网络的结构：

$$y_t = f(x_t, x_{t-1})$$

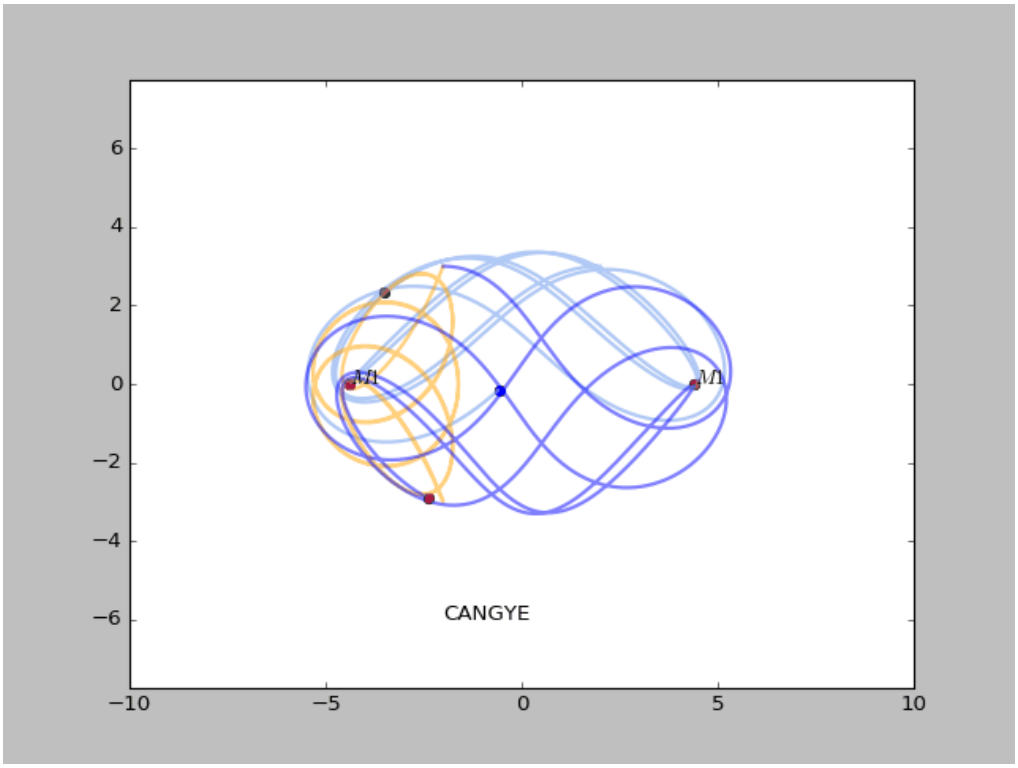
或者说：



方程描述：

$$\frac{dx}{dt} = 40.0(y - x) + 0.16xz$$
$$\frac{dy}{dt} = 55.0x - xz + 20.0y$$
$$\frac{dz}{dt} = 1.833z + xy + 0.65x^2$$

那么数值求解上述方程的过程实际上就是类似于RNN的过程。



很多时候其表现出概率的特征，这是可以通过统计分析得到的。

有兴趣的，可以参考数值模拟过程用神经网络重建吸引子。