第四章、神经网络优化

针对《TensorFlow学习笔记》做相关学习笔记，这一段落主要记录神经网络的优化相关的知识。

4.1 正则化的理论知识

4.1.1 过拟合

神经网络模型在训练数据集上表现很好，但是却在新的预测或者分类的数据集上的表现不是很理想，这就说明模型的泛化能力差，可能存在过拟合现象(也有可能是存在欠拟合的情况)

4.1.2 正则化

在损失函数中给每个参数加上权重参数ω加上权重系数η,引入模型复杂度指标，实现对模型的噪声抑制，避免最终的模型存在过拟合现象。

4.1.3 正则化的理论思路

机器学习的大部分带参模型的结构基本上和如下模型形似，模型如下：

$$\omega^* = arg\min_{\omega} \sum_i L(y_i, f(x_i;\omega)) + \alpha\Omega(\omega)$$

其中α为正则化系数, 也是一个权值。Ω是一个规则化函数。

- [1] 规则化函数Ω

规则化函数Ω, 有很多种选择，一般是性复杂度的单调递增函数,模型越复杂，规则化值就越大。一般常见的比如L0范数, L1范数，迹范数，Frobenius范数和核范数等等。

- [2] 正则化的目标函数

$$\widetilde{J}(\theta;X,y) = J(\theta;X,y) + \alpha\Omega(\theta)$$

$\widetilde{J}$为正则化后的函数；

$J(\theta;X,y)$为标准目标函数；

Ω是权衡范数惩罚项；

$\alpha \in [0,\propto)$是权衡范数惩罚项Ω和标准目标函数$J(\theta;X,y)$的相对贡献超参数；

注解1:分类和回归问题的区别

输入变量与输出变量均为连续变量的预测问题是回归问题；

输出变量为有限个离散变量的预测问题成为分类问题；

Logistic回归，也可以说是二分类的情况；

4.1.4 L^2参数正则化

L_2范数可以防止过拟合，提升模型的泛化能力。

$$\Omega(\theta) = \frac{1}{2}||\omega||_2^2$$

为了简单的表示，我们假设不考虑偏置,模型中只存在权重系数ω ,则$\theta == \omega$,代入模型的总的目标函数，得到如下表达式。

$$\widetilde{J}(\omega;X,y) = J(\omega;X,y) + \alpha\frac{1}{2}||\omega||_2^2$$
$$= J(\omega;X,y) + \frac{\alpha}{2}\omega^\top\omega$$

与之对应的梯度为：

$$\nabla_\omega\widetilde{J}(\omega;X,y) = \alpha\omega + \nabla_\omega J(\omega;X,y)$$

使用单步梯度下降更新权重，即执行如下更新：

$$\omega \leftarrow \omega - \epsilon(\alpha\omega + \nabla_\omega J(\omega;X,y))$$

这种写法对上面的进一步改写就是这样的

$$\omega \leftarrow \omega - \epsilon\nabla_\omega\widetilde{J}(\omega;X,y)$$

有没有发现 ϵ和梯度下降算法的学习率表达式几乎一致。

4.1.5 L^1参数正则化

类似，对模型参数ω的L^1正则化被定义为：

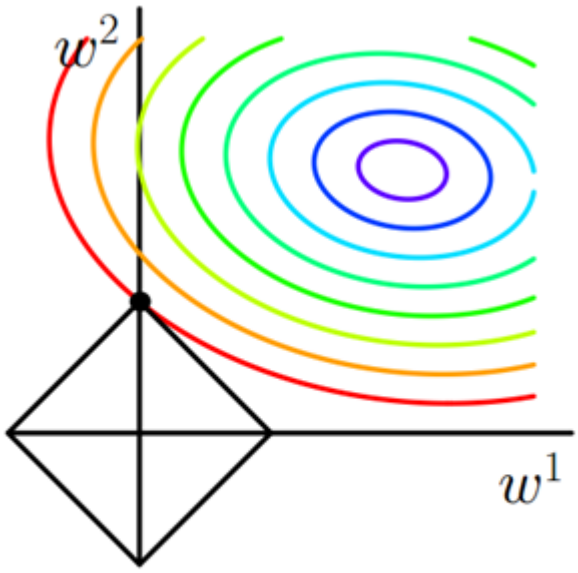
$$\Omega(\theta) = ||\omega||_1 = \sum_i |\omega_i|$$

对应的正则化目标函数为：

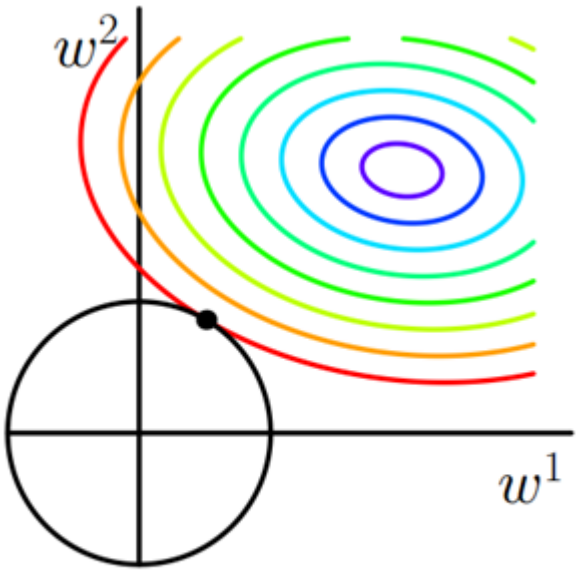
$$\widetilde{J}(\omega;X,y) = \alpha||\omega||_1 + J(\omega;X,y)$$

4.1.6 L^2、L^2参数正则化的区别

为了便于可视化，我们考虑两维的情况，在(w1, w2)平面上可以画出目标函数的等高线，而约束条件则成为平面上半径为C的一个 norm ball 。等高线与 norm ball 首次相交的地方就是最优解，如下图：



(a) ℓ_1-ball meets quadratic function. ℓ_1-ball has corners. It's very likely that the meet-point is at one of the corners.



(b) ℓ_2-ball meets quadratic function. ℓ_2-ball has no corner. It is very unlikely that the meet-point is on any of axes.

通过这个$L_1 - ball$和$L_2 - ball$图像可以看出，L_1和每隔坐标轴都有"角"的出现，最优解如果出现在轴上，代表对应的轴上的ω参数为0，例如图中的相交点就有w1=0，而更高维的时候（想象一下三维的L1-ball 是什么样的？）除了角点以外，还有很多边的轮廓也是既有很大的概率成为第一次相交的地方，又会产生稀疏性。

相比之下，L2-ball 就没有这样的性质，因为没有角，所以第一次相交的地方出现在具有稀疏性的位置的概率就变得非常小了。这就从直观上来解释了为什么L1-regularization 能产生稀疏性，而L2-

regularization 不行的原因了。

注解1:批量梯度下降算法

批量梯度下降算法(batch gradient descent)的公式为：

repeat until convergence{

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial_j} J(\theta_0, \theta_1) \qquad (for j = 0 \quad and \quad j = 1)$$

}

α就是学习率, 他决定了代价函数沿着梯度下降程度最大的方向向下迈出的下一步的步长；

其中代价函数的梯度代表下一步迈步的方向；

前面一个负号，代表，方向永远向着局部梯度最优点的方向；

参考来自：

- 机器学习中的范数规则化之（一）L0、L1与L2范数
- 小花书

4.2 正则化的代码实现

4.2.1 基础知识

通过对未经正则化前的散点图和经过正则化后的散点图进行对比，我们可以发现，如果引入正则化后，所能够带来的优势。

4.2.1 TensorFlow基本函数

- tf.add_to_collection('list_name',element)

将元素element添加到列表list_name中。

- tf.get_collection('list_name')

返回名称为list_name的列表

- tf.add_n(list)

将列表元素相加并返回

- tf.train.AdamOptimizer()

Adam 这个名字来源于 **adaptive moment estimation** , 自适应矩估计，如果一个随机变量X服从某个分布，X的一阶矩是求取X样本的平均值，表示为$E(X)$。X的二阶矩是表示求取样本的平方的平均值，表示为$E(X^2)$

Adam 也是基于梯度下降的方法，但是每次迭代参数的学习步长都有一个确定的范围，不会因为很大的梯度导致很大的学习步长，参数的值比较稳定，AdamOptimizer通过动量（参数的移动平均数）来改善传统梯度下降，促进超参数动态调整。

```
import tensorflow as tf
tf.add_to_collection('losses', tf.constant(2.2))
tf.add_to_collection('losses', tf.constant(3.))
with tf.Session() as sess:
    print(sess.run(tf.get_collection('losses')))
    print(sess.run(tf.add_n(tf.get_collection('losses'))
```

结果：
[2.2, 3.0]
5.2
注意：
使用tf.add_n对列表元素进行相加时，列表内元素类型必须一致，否则会报错。

```
#coding:utf-8
# 引入正则化,精化模型
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

BATCH_SIZE = 30
seed = 20

rdm = np.random.RandomState(seed)
X = rdm.randn(300, 2)

Y_ = [int(x0*x0+x1*x1 < 2) for (x0, x1) in X]
Y_c = [['red' if y else 'blue'] for y in Y_]

X = np.vstack(X).reshape(-1, 2)
Y_ = np.vstack(Y_).reshape(-1, 1)

# print(X)
# print(Y_)
# print(Y_c)

plt.scatter(X[:, 0], X[:, 1], c=np.squeeze(Y_c))
plt.show()

# 定义前向传播
def get_weight(shape, regularizer):
    w=tf.Variable(tf.random_normal(shape), dtype=tf.float32)
    tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularizer)(w))
    return w


def get_bias(shape):
    b = tf.Variable(tf.constant(0.01, shape=shape))
    return b


x = tf.placeholder(tf.float32, shape=(None, 2))
y_ = tf.placeholder(tf.float32, shape=(None, 1))

w1 = get_weight([2, 11], 0.01)
b1 = get_bias([11])
y1 = tf.nn.relu(tf.matmul(x, w1)+b1)

w2 = get_weight([11, 1], 0.01)
b2 = get_bias([1])
y = tf.matmul(y1, w2) + b2


# 定义损失函数
loss_mse = tf.reduce_mean(tf.square(y-y_))
# tf.add_n([p1,p2,p3,...]) 函数是实现一个列表的元素相加
loss_total = loss_mse + tf.add_n(tf.get_collection('losses'))

# 定义反向传播
train_step = tf.train.AdamOptimizer(0.0001).minimize(loss_mse)
# train_step = tf.train.GradientDescentOptimizer(0.001).minimize(loss_mse)


with tf.Session() as sess:
    init_op = tf.global_variables_initializer()
    sess.run(init_op)
    STEPS = 40000
```

```
    for i in range(STEPS):
        start = (i*BATCH_SIZE) % 300
        end = start + BATCH_SIZE
        sess.run(train_step, feed_dict={x:X[start:end], y_:Y_[start:end]})
        if i%2000 == 0:
            loss_mse_val = sess.run(loss_mse, feed_dict={x:X, y_:Y_})# 这地方我可以
            print("After %d steps, loss is:%f"%(i, loss_mse_val))

    xx, yy = np.mgrid[-3:3:.01, -3:3:.01]
    grid = np.c_[xx.ravel(), yy.ravel()]
    probs = sess.run(y, feed_dict={x:grid})

    probs = probs.reshape(xx.shape)
    print("w1:\n",sess.run(w1))
    print("b1:\n",sess.run(b1))
    print("w2:\n",sess.run(w2))
    print("b2:\n",sess.run(b2))
plt.scatter(X[:,0], X[:,1],c=np.squeeze(Y_c))
plt.contour(xx, yy, probs, levels=[.5])
plt.show()

train_step = tf.train.AdamOptimizer(0.0001).minimize(loss_total)

with tf.Session() as sess:
    init_op = tf.global_variables_initializer()
    sess.run(init_op)
    STEPS = 40000
    for i in range(STEPS):
        start = (i*BATCH_SIZE) % 300
        end = start + BATCH_SIZE
        sess.run(train_step, feed_dict={x: X[start:end], y_:Y_[start:end]})
        if i % 2000 == 0:
            loss_v = sess.run(loss_total, feed_dict={x:X,y_:Y_})
            print("After %d steps, loss is: %f" %(i, loss_v))

    xx, yy = np.mgrid[-3:3:.01, -3:3:.01]
    grid = np.c_[xx.ravel(), yy.ravel()]
    probs = sess.run(y, feed_dict={x:grid})
    probs = probs.reshape(xx.shape)
    print("w1:\n",sess.run(w1))
    print("b1:\n",sess.run(b1))
    print("w2:\n",sess.run(w2))
    print("b2:\n",sess.run(b2))

plt.scatter(X[:,0], X[:,1], c=np.squeeze(Y_c))
plt.contour(xx, yy, probs, levels=[.5])
plt.show()
```

4.3 综合代码

包括了正则化，包括指数衰减学习率

```
#coding:utf-8
#0导入模块 ，生成模拟数据集
import numpy as np
import matplotlib.pyplot as plt
seed = 2
```

```
def generateds():
    #基于seed产生随机数
    rdm = np.random.RandomState(seed)
    #随机数返回300行2列的矩阵，表示300组坐标点（x0,x1）作为输入数据集
    X = rdm.randn(300,2)
    #从X这个300行2列的矩阵中取出一行,判断如果两个坐标的平方和小于2，给Y赋值1，其余赋值0
    #作为输入数据集的标签（正确答案）
    Y_ = [int(x0*x0 + x1*x1 <2) for (x0,x1) in X]
    #遍历Y中的每个元素，1赋值'red'其余赋值'blue'，这样可视化显示时人可以直观区分
    Y_c = [['red' if y else 'blue'] for y in Y_]
    #对数据集X和标签Y进行形状整理，第一个元素为-1表示跟随第二列计算，第二个元素表示多少列，可见
    X = np.vstack(X).reshape(-1,2)
    Y_ = np.vstack(Y_).reshape(-1,1)

    return X, Y_, Y_c

#print X
#print Y_
#print Y_c
#用plt.scatter画出数据集X各行中第0列元素和第1列元素的点即各行的（x0，x1），用各行Y_c对应的值
#plt.scatter(X[:,0], X[:,1], c=np.squeeze(Y_c))
#plt.show()

#定义神经网络的输入、参数和输出，定义前向传播过程
def get_weight(shape, regularizer):
    w = tf.Variable(tf.random_normal(shape), dtype=tf.float32)
    tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularizer)(w))
    return w

def get_bias(shape):
    b = tf.Variable(tf.constant(0.01, shape=shape))
    return b

def forward(x, regularizer):

    w1 = get_weight([2,11], regularizer)
    b1 = get_bias([11])
    y1 = tf.nn.relu(tf.matmul(x, w1) + b1)

    w2 = get_weight([11,1], regularizer)
    b2 = get_bias([1])
    y = tf.matmul(y1, w2) + b2

    return y
```

```
#coding:utf-8
#0导入模块 ，生成模拟数据集
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
# import opt4_8_generateds
# import opt4_8_forward

STEPS = 40000
BATCH_SIZE = 30
LEARNING_RATE_BASE = 0.001
LEARNING_RATE_DECAY = 0.999
REGULARIZER = 0.01

def backward():
    x = tf.placeholder(tf.float32, shape=(None, 2))
    y_ = tf.placeholder(tf.float32, shape=(None, 1))

    X, Y_, Y_c = generateds()

    y = forward(x, REGULARIZER)

    global_step = tf.Variable(0,trainable=False)

    learning_rate = tf.train.exponential_decay(
        LEARNING_RATE_BASE,
        global_step,
```

```
        300/BATCH_SIZE,
        LEARNING_RATE_DECAY,
        staircase=True)


    #定义损失函数
    loss_mse = tf.reduce_mean(tf.square(y-y_))
    loss_total = loss_mse + tf.add_n(tf.get_collection('losses'))

    #定义反向传播方法：包含正则化
    train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss_total)

    with tf.Session() as sess:
        init_op = tf.global_variables_initializer()
        sess.run(init_op)
        for i in range(STEPS):
            start = (i*BATCH_SIZE) % 300
            end = start + BATCH_SIZE
            sess.run(train_step, feed_dict={x: X[start:end], y_:Y_[start:end]})
            if i % 2000 == 0:
                loss_v = sess.run(loss_total, feed_dict={x:X,y_:Y_})
                print("After %d steps, loss is: %f" %(i, loss_v))

        xx, yy = np.mgrid[-3:3:.01, -3:3:.01]
        grid = np.c_[xx.ravel(), yy.ravel()]
        probs = sess.run(y, feed_dict={x:grid})
        probs = probs.reshape(xx.shape)

    plt.scatter(X[:,0], X[:,1], c=np.squeeze(Y_c))
    plt.contour(xx, yy, probs, levels=[.5])
    plt.show()

if __name__=='__main__':
    backward()
```