暑假学习记录

8.3(deeplearing 中的优化算法)

mini-batch梯度下降法

在机器学习中，最简单就是没有任何优化的梯度下降(GD,Gradient Descent)，我们每一次循环都是对整个训练集进行学习，这叫做批量梯度下降(Batch Gradient Descent)
由梯度下降算法演变来的还有随机梯度下降（SGD）算法和小批量梯度下降算法，随机梯度下降（SGD），相当于小批量梯度下降，但是和mini-batch不同的是其中每个小批量(mini-batch)仅有1个样本，和梯度下降不同的是你一次只能在一个训练样本上计算梯度，而不是在整个训练集上计算梯度。
我们来看一下他的差异

```
#仅做比较，不运行。

#批量梯度下降，又叫梯度下降
X = data_input
Y = labels

parameters = initialize_parameters(layers_dims)
for i in range(0,num_iterations):
    #前向传播
    A,cache = forward_propagation(X,parameters)
    #计算损失
    cost = compute_cost(A,Y)
    #反向传播
    grads = backward_propagation(X,Y,cache)
    #更新参数
    parameters = update_parameters(parameters,grads)

#随机梯度下降算法：
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in (0,num_iterations):
    for j in m:
        #前向传播
        A,cache = forward_propagation(X,parameters)
        #计算成本
        cost = compute_cost(A,Y)
        #后向传播
        grads = backward_propagation(X,Y,cache)
        #更新参数
        parameters = update_parameters(parameters,grads)
```

在随机梯度下降中，在更新梯度之前，只使用1个训练样本。 当训练集较大时，随机梯度下降可以更快，但是参数会向最小值摆动，而不是平稳地收敛.
在实际中，更好的方法是使用小批量(mini-batch)梯度下降法，小批量梯度下降法是一种综合了梯度下降法和随机梯度下降法的方法，在它的每次迭代中，既不是选择全部的数据来学习，也不是选择一个样本来学习，而是把所有的数据集分割为一小块一小块的来学习，它会随机选择一小块（mini-batch），块大小一般为2的n次方倍。一方面，充分利用的GPU的并行性，更一方面，不会让计算时间特别长。
我们要使用mini-batch要经过两个步骤：

1. 把训练集打乱，但是X和Y依旧是一一对应的，之后，X的第i列是与Y中的第i个标签对应的样本。乱序步骤确保将样本被随机分成不同的小批次。
2. 切分，我们把训练集打乱之后，我们就可以对它进行切分了。 首先来获取mini-batch

```
def random_mini_batches(X,Y,mini_batch_size=64,seed=0):
    """
    从（X，Y）中创建一个随机的mini-batch列表

    参数：
        X - 输入数据，维度为(输入节点数量，样本的数量)
        Y - 对应的是X的标签，【1 | 0】（蓝|红），维度为(1,样本的数量)
        mini_batch_size - 每个mini-batch的样本数量

    返回：
        mini-bacthes - 一个同步列表，维度为（mini_batch_X,mini_batch_Y）

    """

    np.random.seed(seed) #指定随机种子
    m = X.shape[1]
    mini_batches = []

    #第一步：打乱顺序
    permutation = list(np.random.permutation(m)) #它会返回一个长度为m的随机数组，且里面的数是0到m-1
    shuffled_X = X[:,permutation]   #将每一列的数据按permutation的顺序来重新排列。
    shuffled_Y = Y[:,permutation].reshape((1,m))

    """
    #博主注：
    #如果你不好理解的话请看一下下面的伪代码，看看X和Y是如何根据permutation来打乱顺序的。
    x = np.array([[1,2,3,4,5,6,7,8,9],
                  [9,8,7,6,5,4,3,2,1]])
    y = np.array([[1,0,1,0,1,0,1,0,1]])

    random_mini_batches(x,y)
    permutation= [7, 2, 1, 4, 8, 6, 3, 0, 5]
    shuffled_X= [[8 3 2 5 9 7 4 1 6]
                 [2 7 8 5 1 3 6 9 4]]
    shuffled_Y= [[0 1 0 1 1 1 0 1 0]]
    """
```

```
    #第二步，分割
    num_complete_minibatches = math.floor(m / mini_batch_size) #把你的训练集
分割成多少份,请注意，如果值是99.99，那么返回值是99，剩下的0.99会被舍弃
    for k in range(0,num_complete_minibatches):
        mini_batch_X = shuffled_X[:,k * mini_batch_size:
(k+1)*mini_batch_size]
        mini_batch_Y = shuffled_Y[:,k * mini_batch_size:
(k+1)*mini_batch_size]
        """
        #博主注：
        #如果你不好理解的话请单独执行下面的代码，它可以帮你理解一些。
        a = np.array([[1,2,3,4,5,6,7,8,9],
                      [9,8,7,6,5,4,3,2,1],
                      [1,2,3,4,5,6,7,8,9]])
        k=1
        mini_batch_size=3
        print(a[:,1*3:(1+1)*3]) #从第4列到第6列
        '''
        [[4 5 6]
         [6 5 4]
         [4 5 6]]
        '''
        k=2
        print(a[:,2*3:(2+1)*3]) #从第7列到第9列
        '''
        [[7 8 9]
         [3 2 1]
         [7 8 9]]
        '''

        #看一下每一列的数据你可能就会好理解一些
        """
        mini_batch = (mini_batch_X,mini_batch_Y)
        mini_batches.append(mini_batch)

    #如果训练集的大小刚好是mini_batch_size的整数倍，那么这里已经处理完了
    #如果训练集的大小不是mini_batch_size的整数倍，那么最后肯定会剩下一些，我们要把它处
理了
    if m % mini_batch_size != 0:
        #获取最后剩余的部分
        mini_batch_X = shuffled_X[:,mini_batch_size *
num_complete_minibatches:]
        mini_batch_Y = shuffled_Y[:,mini_batch_size *
num_complete_minibatches:]

        mini_batch = (mini_batch_X,mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

我们先来看看分割后如何获取第一第二个mini-batch吧~

```
#第一个mini-batch
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
#第二个mini-batch
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 * mini_batch_size]
```

包含动量的梯度下降

由于小批量梯度下降只看到了一个子集的参数更新，更新的方向有一定的差异，所以小批量梯度下降的路径将“振荡地”走向收敛，使用动量可以减少这些振荡，动量考虑了过去的梯度以平滑更新， 我们将把以前梯度的方向存储在变量v中，从形式上讲，这将是前面的梯度的指数加权平均值。

代码实现：

```
for l in range(L)://L为层数
        #计算速度
        v["dW" + str(l + 1)] = beta * v["dW" + str(l + 1)] + (1 - beta) *
grads["dW" + str(l + 1)]
        v["db" + str(l + 1)] = beta * v["db" + str(l + 1)] + (1 - beta) *
grads["db" + str(l + 1)]

        #更新参数
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] -
learning_rate * v["dW" + str(l + 1)]
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] -
learning_rate * v["db" + str(l + 1)]
```

需要注意的是速度v v v是用0来初始化的，因此，该算法需要经过几次迭代才能把速度提升上来并开始跨越更大步伐。当beta=0时，该算法相当于是没有使用momentum算法的标准的梯度下降算法。当beta越大的时候，说明平滑的作用越明显。通常0.9是比较合适的值。

Adam算法

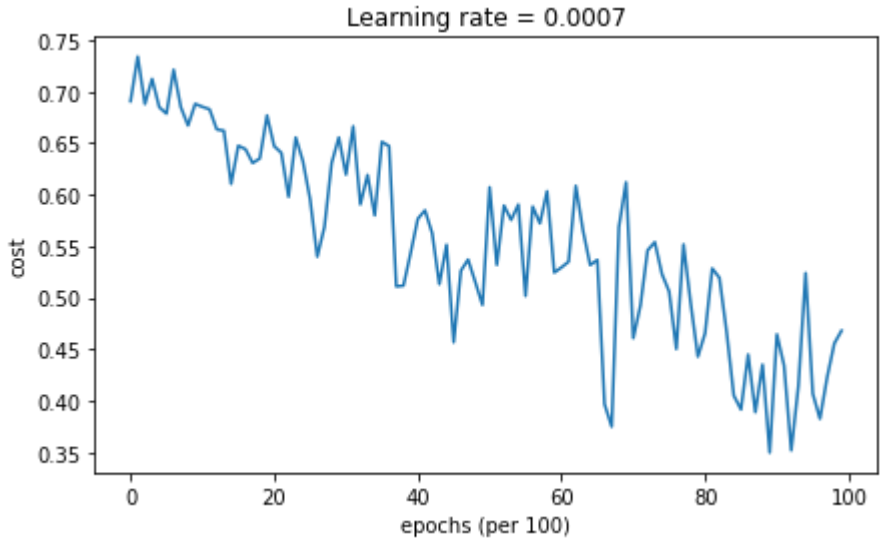
Adam算法是训练神经网络中最有效的算法之一，它是RMSProp算法与Momentum算法的结合体。我们来看看它都干了些什么吧~

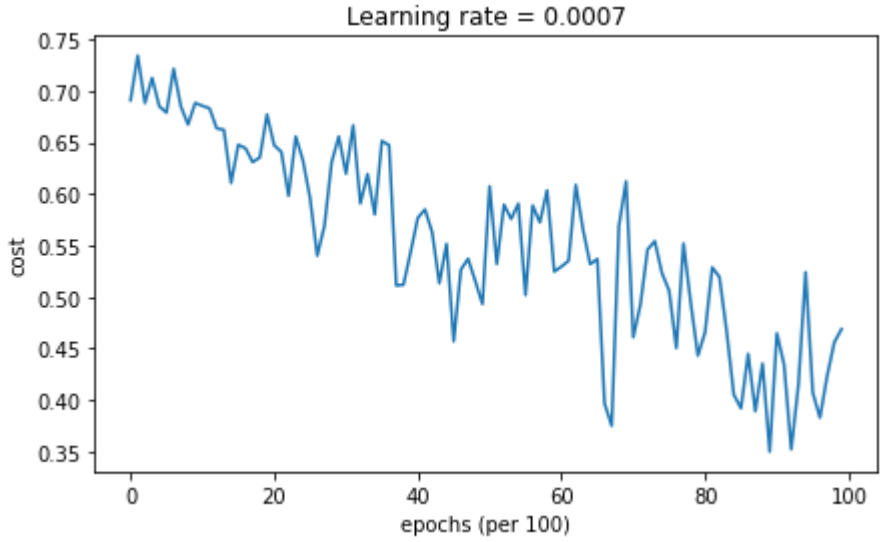
1. 计算以前的梯度的指数加权平均值
2. 计算以前梯度的平方的指数加权平均值
3. 根据1和2更新参数。 代码实现：

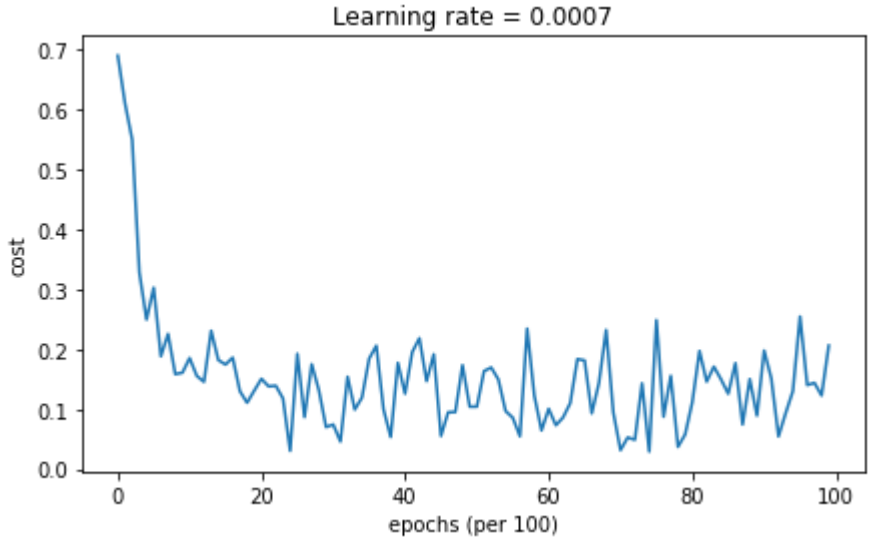
```
for l in range(L):
        #梯度的移动平均值,输入："v , grads , beta1",输出：" v "
        v["dW" + str(l + 1)] = beta1 * v["dW" + str(l + 1)] + (1 - beta1) *
grads["dW" + str(l + 1)]
        v["db" + str(l + 1)] = beta1 * v["db" + str(l + 1)] + (1 - beta1) *
grads["db" + str(l + 1)]

        #计算第一阶段的偏差修正后的估计值，输入"v , beta1 , t" , 输
```

```
出："v_corrected"
        v_corrected["dW" + str(l + 1)] = v["dW" + str(l + 1)] / (1 -
np.power(beta1,t))
        v_corrected["db" + str(l + 1)] = v["db" + str(l + 1)] / (1 -
np.power(beta1,t))

        #计算平方梯度的移动平均值，输入："s, grads , beta2"，输出："s"
        s["dW" + str(l + 1)] = beta2 * s["dW" + str(l + 1)] + (1 - beta2) *
np.square(grads["dW" + str(l + 1)])
        s["db" + str(l + 1)] = beta2 * s["db" + str(l + 1)] + (1 - beta2) *
np.square(grads["db" + str(l + 1)])

        #计算第二阶段的偏差修正后的估计值，输入："s , beta2 , t"，输
出："s_corrected"
        s_corrected["dW" + str(l + 1)] = s["dW" + str(l + 1)] / (1 -
np.power(beta2,t))
        s_corrected["db" + str(l + 1)] = s["db" + str(l + 1)] / (1 -
np.power(beta2,t))

        #更新参数，输入: "parameters, learning_rate, v_corrected,
s_corrected, epsilon". 输出: "parameters".
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] -
learning_rate * (v_corrected["dW" + str(l + 1)] / np.sqrt(s_corrected["dW"
+ str(l + 1)] + epsilon))
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] -
learning_rate * (v_corrected["db" + str(l + 1)] / np.sqrt(s_corrected["db"
+ str(l + 1)] + epsilon))
```

我们可以对比一下没有任何优化的梯度下降 具有动量的梯度下降测试 Adam优化后的梯度下降测试







我们可以看到前两者大致相同，振幅比较明显，总体成下降趋势。而具有动量的梯度下降测试是因为这个例子比较简单，使用动量效果很小，但对于更复杂的问题，你可能会看到更好的效果。
而Adam优化后的梯度下降测试就比较平滑，下降较快。效果较好。

8.5(Tensorflow 入门)

导入Tensorflow

import tensorflow.compat.v1 as tf

其中，「compat」是 TF2.X 专门为兼容 TF 1.X 配置的模块。目前，还是有很多前沿研究，放不下 TF 1.X。那就更不用说之前的经典模型，绝大多都是 TF 1.X 写的。

```
y_hat = tf.constant(36,name="y_hat")            #定义y_hat为固定值36
y = tf.constant(39,name="y")                    #定义y为固定值39

loss = tf.Variable((y-y_hat)**2,name="loss" )    #为损失函数创建一个变量

init = tf.global_variables_initializer()        #运行之后的初始化
(session.run(init))
                                                #损失变量将被初始化并准备计算
with tf.Session() as session:                   #创建一个session并打印输出
    session.run(init)                           #初始化变量
```

```
    print(session.run(loss))                    #打印损失值
```

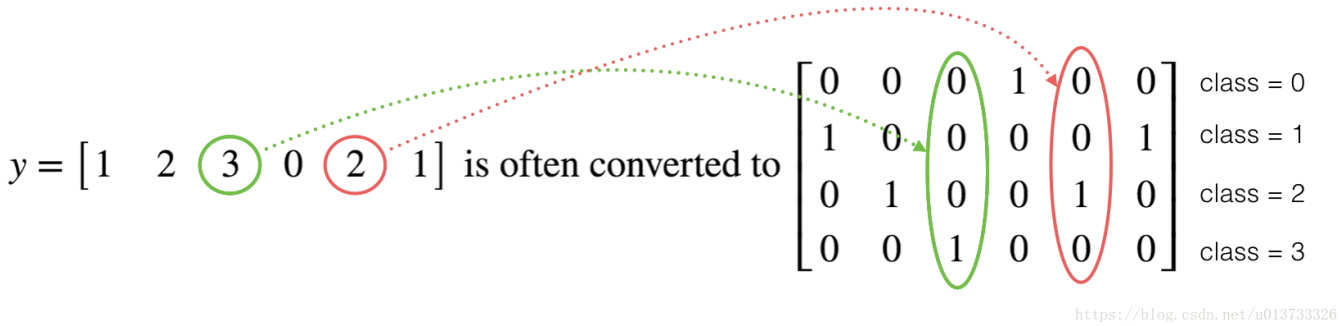
对于Tensorflow的代码实现而言，实现代码的结构如下：

1. 创建Tensorflow变量（此时，尚未直接计算）
2. 实现Tensorflow变量之间的操作定义
3. 初始化Tensorflow变量
4. 创建Session
5. 运行Session，此时，之前编写操作都会在这一步运行。 在启动图时（进行操作之前），所有的变量必须被明确定义。变量常用来储存和更新参数，在计算图过程中其值会一直保存至程序运行结束，这点正是区别于一般的张量。一般的Tensorflow张量在运行过程中仅仅是从计算图中流过，并不会被保存下来。涉及到变量的相关操作必须通过session会话控制。
   特别注意 在tensorflow的世界里变量的定义和初始化是被分开的。
   tf.global_variables_initializer()用于初始化所有变量；w.initializer用于初始化单个变量。
   实质上，tf.Variable()是真正的定义变量
   一个简单的成本函数

```
coefficients = np.array([[1.],[-10.],[25.]])


w = tf.Variable(0,dtype=tf.float32)
x = tf.placeholder(tf.float32,[3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
for i in range(1000):
    session.run(train,feed_dict={x:coefficients})

print(session.run(w))
```

从结果可以看到，w如预期的非常接近5.

使用独热编码（0、1编码）
很多时候在深度学习中 y y y向量的维度是从0到 C – 1 的， C是指分类的类别数量，如果 C = 4 那么对 y而言你可能需要有以下的转换方式:



要在numpy中进行这种转换，您可能需要编写几行代码。 在tensorflow中，只需要使用一行代码：

```
tf.one_hot(labels,depth,axis)
```

我们可以写一个函数来实现独热编码

```
def one_hot_matrix(lables,C):
    """
    创建一个矩阵，其中第i行对应第i个类号，第j列对应第j个训练样本
    所以如果第j个样本对应着第i个标签，那么entry (i,j)将会是1

    参数：
        lables - 标签向量
        C - 分类数

    返回：
        one_hot - 独热矩阵

    """

    #创建一个tf.constant，赋值为C，名字叫C
    C = tf.constant(C,name="C")

    #使用tf.one_hot，注意一下axis
    one_hot_matrix = tf.one_hot(indices=lables , depth=C , axis=0)

    #创建一个session
    sess = tf.Session()

    #运行session
    one_hot = sess.run(one_hot_matrix)

    #关闭session
    sess.close()

    return one_hot
```

使用Tensorflow构建一个神经网络

用Tensorfolw我们只用关注向前传播，而向后传播和更新参数在Tensorflow可以自动完成。 在这里为了篇幅我

就不放上向前传播的代码了。 **成本函数**

```
def compute_cost(Z3,Y):
    """
    计算成本

    参数：
        Z3 - 前向传播的结果
        Y - 标签，一个占位符，和Z3的维度相同

    返回：
        cost - 成本值


    """
    logits = tf.transpose(Z3) #转置
    labels = tf.transpose(Y)  #转置

    cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,labels
=labels))#tf.reduce_mean()是对其求平均，相当于/m

    return cost
```

得益于编程框架，所有反向传播和参数更新都在1行代码中处理。计算成本函数后，将创建一个“optimizer”对象。 运行tf.session时，必须将此对象与成本函数一起调用，当被调用时，它将使用所选择的方法和学习速率对给定成本进行优化。
对于梯度下降：

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate =
learning_rate).minimize(cost)
```

要进行优化，应该这样做：

```
_ , c = sess.run([optimizer,cost],feed_dict=
{X:mini_batch_X,Y:mini_batch_Y})
#等价于
#sess.run(optimizer,feed_dict={X：mini_batch_X,Y:mini_batch_Y})
#c = sess.run(cost,feed_dict={X:mini_batch_X,Y:mini_batch_Y})
```

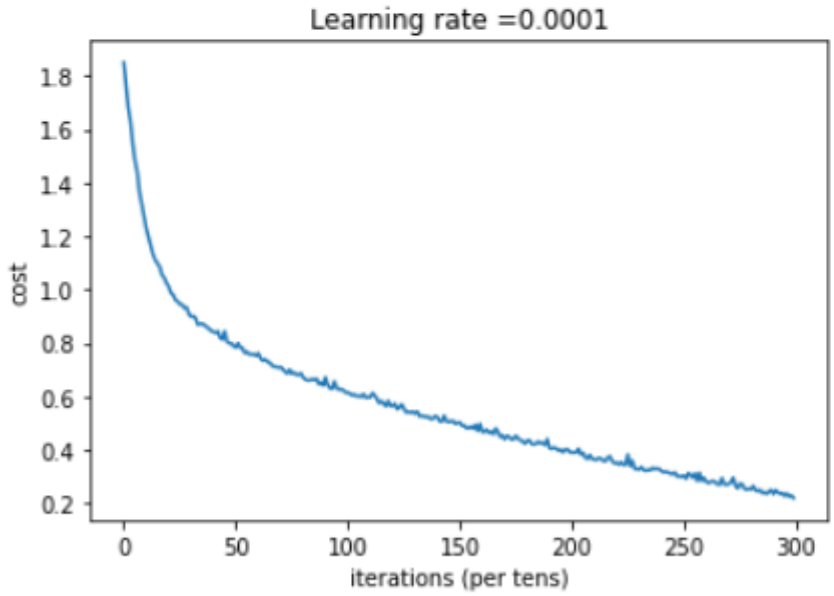
现在我们将实现我们的模型:

```
def model(X_train,Y_train,X_test,Y_test,
        learning_rate=0.0001,num_epochs=1500,minibatch_size=32,
        print_cost=True,is_plot=True):
    """
```

```
    实现一个三层的TensorFlow神经网络：LINEAR->RELU->LINEAR->RELU->LINEAR-
>SOFTMAX

    参数：
        X_train - 训练集，维度为（输入大小（输入节点数量） = 12288, 样本数量 =
1080）
        Y_train - 训练集分类数量，维度为（输出大小(输出节点数量) = 6, 样本数量 =
1080）
        X_test - 测试集，维度为（输入大小（输入节点数量） = 12288, 样本数量 = 120）
        Y_test - 测试集分类数量，维度为（输出大小(输出节点数量) = 6, 样本数量 =
120）
        learning_rate - 学习速率
        num_epochs - 整个训练集的遍历次数
        mini_batch_size - 每个小批量数据集的大小
        print_cost - 是否打印成本，每100代打印一次
        is_plot - 是否绘制曲线图

    返回：
        parameters - 学习后的参数

    """
    ops.reset_default_graph()                     #能够重新运行模型而不覆盖tf变量
    tf.set_random_seed(1)
    seed = 3
    (n_x , m)  = X_train.shape                    #获取输入节点数量和样本数
    n_y = Y_train.shape[0]                        #获取输出节点数量
    costs = []                                    #成本集

    #给X和Y创建placeholder
    X,Y = create_placeholders(n_x,n_y)

    #初始化参数
    parameters = initialize_parameters()

    #前向传播
    Z3 = forward_propagation(X,parameters)

    #计算成本
    cost = compute_cost(Z3,Y)

    #反向传播，使用Adam优化
    optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

    #初始化所有的变量
    init = tf.global_variables_initializer()

    #开始会话并计算
    with tf.Session() as sess:
        #初始化
        sess.run(init)

        #正常训练的循环
        for epoch in range(num_epochs):
```

```
            epoch_cost = 0  #每代的成本
            num_minibatches = int(m / minibatch_size)    #minibatch的总数量
            seed = seed + 1
            minibatches =
tf_utils.random_mini_batches(X_train,Y_train,minibatch_size,seed)

            for minibatch in minibatches:

                #选择一个minibatch
                (minibatch_X,minibatch_Y) = minibatch

                #数据已经准备好了，开始运行session
                _ , minibatch_cost = sess.run([optimizer,cost],feed_dict=
{X:minibatch_X,Y:minibatch_Y})

                #计算这个minibatch在这一代中所占的误差
                epoch_cost = epoch_cost + minibatch_cost / num_minibatches

            #记录并打印成本
            ## 记录成本
            if epoch % 5 == 0:
                costs.append(epoch_cost)
                #是否打印：
                if print_cost and epoch % 100 == 0:
                        print("epoch = " + str(epoch) + "    epoch_cost = "
+ str(epoch_cost))

        #是否绘制图谱
        if is_plot:
            plt.plot(np.squeeze(costs))
            plt.ylabel('cost')
            plt.xlabel('iterations (per tens)')
            plt.title("Learning rate =" + str(learning_rate))
            plt.show()

        #保存学习后的参数
        parameters = sess.run(parameters)
        print("参数已经保存到session。")

        #计算当前的预测结果
        correct_prediction = tf.equal(tf.argmax(Z3),tf.argmax(Y))

        #计算准确率
        accuracy = tf.reduce_mean(tf.cast(correct_prediction,"float"))

        print("训练集的准确率：", accuracy.eval({X: X_train, Y: Y_train}))
        print("测试集的准确率:", accuracy.eval({X: X_test, Y: Y_test}))

        return parameters
```

```
epoch = 0    epoch_cost = 1.8509137196974321
epoch = 100    epoch_cost = 1.0115901141455679
epoch = 200    epoch_cost = 0.8412949713793667
epoch = 300    epoch_cost = 0.7613892952601116
epoch = 400    epoch_cost = 0.6715726400866652
epoch = 500    epoch_cost = 0.6138875610900649
epoch = 600    epoch_cost = 0.5649370876225559
epoch = 700    epoch_cost = 0.5216509997844696
epoch = 800    epoch_cost = 0.46676008448456274
epoch = 900    epoch_cost = 0.43284833250623755
epoch = 1000    epoch_cost = 0.3891014619307084
epoch = 1100    epoch_cost = 0.3510727579846527
epoch = 1200    epoch_cost = 0.3201399996425166
epoch = 1300    epoch_cost = 0.2798012142831629
epoch = 1400    epoch_cost = 0.2519830131169521
```




```
参数已经保存到session。
训练集的准确率:  0.9546296
测试集的准确率: 0.7
CPU的执行时间 = 366.2936760000001 秒
```