暑假学习记录

8.13

正交化

正交化的概念是指，你可以想出一个维度，这个维度你想做的是控制转向角，还有另一个维度来控制你的速度，那么你就需要一个旋钮尽量只控制转向角，另一个旋钮，在这个开车的例子里其实是油门和刹车控制了你的速度。但如果你有一个控制旋钮将两者混在一起，比如说这样一个控制装置同时影响你的转向角和速度，同时改变了两个性质，那么就很难令你的车子以想要的速度和角度前进。然而正交化之后，正交意味着互成 90 度。设计出正交化的控制装置，最理想的情况是和你实际想控制的性质一致，这样你调整参数时就容易得多。可以单独调整转向角，还有你的油门和刹车，令车子以你想要的方式动。
在机器学习中，如果你可以观察你的系统，然后说这一部分是错的，它在训练集上做的不好、在开发集上做的不好、它在测试集上做的不好，或者它在测试集上做的不错，但在现实世界中不好，这就很好。必须弄清楚到底是什么地方出问题了，然后我们刚好有对应的旋钮，或者一组对应的旋钮，刚好可以解决那个问题，那个限制了机器学习系统性能的问题。

单一数字评估

无论你是调整超参数，或者是尝试不同的学习算法，或者在搭建机器学习系统时尝试不同手段，你会发现，如果你有一个单实数评估指标，你的进展会快得多，它可以快速告诉你，新尝试的手段比之前的手段好还是差。如果你要考虑N个指标，有时候选择其中一个指标做为优化指标是合理的。所以你想尽量优化那个指标，然后剩下$N-1$个指标都是满足指标，意味着只要它们达到一定阈值，例如运行时间快于 100 毫秒，但只要达到一定的阈值，你不在乎它超过那个门槛之后的表现，但它们必须达到这个门槛。

训练/开发/测试集划分

机器学习中的工作流程是，你尝试很多思路，用训练集训练不同的模型，然后使用开发集来评估不同的思路，然后选择一个，然后不断迭代去改善开发集的性能，直到最后你可以得到一个令你满意的成本，然后你再用测试集去评估。
将所有数据随机洗牌，放入开发集和测试集，并且开发集和测试集都来自同一分布，这分布就是你的所有数据混在一起。

可避免偏差

如果贝叶斯错误率是 7.5%。你实际上并不想得到低于该级别的错误率，所以你不会说你的训练错误率是 8%，然后 8%就衡量了例子中的偏差大小。你应该说，可避免偏差可能在 0.5%左右，或者 0.5%是可避免偏差的指标。而这个 2%是方差的指标，所以要减少这个 2%比减少这个 0.5%空间要大得多。而在左边的例子中，这 7%衡量了可避免偏差大小，而 2%衡量了方差大小。所以在左边这个例子里，专注减少可避免偏差可能潜力更大。

8.14(卷积神经网络的基本知识)

卷积神经网络将深度学习应用到计算机视觉中，是深度学习算法中的一种，其隐含层内的卷积核参数共享和层间连接的稀疏性使得卷积神经网络能够以较小的计算量对格点化特征，减少了参数，加快了速度。 卷积神经网络的隐含层包含卷积层、池化层和全连接层。

卷积层

卷积层最主要的就是进行卷积运算，其实就是cv中的卷积核，只不过卷积核的参数都是随机的，后通过反向传播一次次更新，只到最佳效果。

池化层

池化层分为最大池化层和均值池化层。最大池化层其实就是当前位置取卷积核范围内最大的数来代替，而均值池化层则是用均值填入。（这里并没有很懂有什么用，不过实验证明，对卷积神经网络效果很好。）

全连接层

其实就是我们之前写的基础神经网络层，一般都是将池化层的结果进行一维化，这样相比直接用全连接层，会少了很多很多数据，大大提高了数度。

代码示范

创建placeholder

```
def create_placeholders(n_H0, n_W0, n_C0, n_y):
    """
    为session创建占位符

    参数：
        n_H0 - 实数，输入图像的高度
        n_W0 - 实数，输入图像的宽度
        n_C0 - 实数，输入的通道数
        n_y  - 实数，分类数

    输出：
        X - 输入数据的占位符，维度为[None, n_H0, n_W0, n_C0]，类型为"float"
        Y - 输入数据的标签的占位符，维度为[None, n_y]，维度为"float"
    """
    X = tf.placeholder(tf.float32,[None, n_H0, n_W0, n_C0])
    Y = tf.placeholder(tf.float32,[None, n_y])

    return X,Y
```

初始化参数

```
def initialize_parameters():
    """
    初始化权值矩阵，这里我们把权值矩阵硬编码：
    W1 : [4, 4, 3, 8]
    W2 : [2, 2, 8, 16]

    返回：
        包含了tensor类型的W1、W2的字典
```

```
    """
    tf.set_random_seed(1)

    W1 = tf.get_variable("W1",
[4,4,3,8],initializer=tf.contrib.layers.xavier_initializer(seed=0))
    W2 = tf.get_variable("W2",
[2,2,8,16],initializer=tf.contrib.layers.xavier_initializer(seed=0))

    parameters = {"W1": W1,
                  "W2": W2}

    return parameters
```

前向传播

在TensorFlow里面有一些可以直接拿来用的函数：
tf.nn.conv2d(X,W1,strides=[1,s,s,1],padding='SAME') 给定输入 X X X和一组过滤器 W 1 W1 W1，这个函数将会自动使用 W 1 W1 W1来对 X X X进行卷积，第三个输入参数是**[1,s,s,1]**是指对于输入 (m, n_H_prev, n_W_prev, n_C_prev)而言，每次滑动的步伐。
tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME')：给定输入 X X X，该函数将会使用大小为（f,f）以及步伐为(s,s)的窗口对其进行滑动取最大值。
tf.nn.relu(Z1)：计算Z1的ReLU激活。
tf.contrib.layers.flatten(P)：给定一个输入P，此函数将会把每个样本转化成一维的向量，然后返回一个tensor变量，其维度为（batch_size,k）
tf.contrib.layers.fully_connected(F, num_outputs)：给定一个已经一维化了的输入F，此函数将会返回一个由全连接层计算过后的输出，使用tf.contrib.layers.fully_connected(F, num_outputs)的时候，全连接层会自动初始化权值且在你训练模型的时候它也会一直参与，所以当我们初始化参数的时候我们不需要专门去初始化它的权值。

```
def forward_propagation(X,parameters):
    """
    实现前向传播
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN ->
FULLYCONNECTED

    参数：
        X - 输入数据的placeholder，维度为(输入节点数量，样本数量)
        parameters - 包含了“W1”和“W2”的python字典。

    返回：
        Z3 - 最后一个LINEAR节点的输出

    """
    W1 = parameters['W1']
    W2 = parameters['W2']

    #Conv2d ： 步伐：1，填充方式：“SAME”
    Z1 = tf.nn.conv2d(X,W1,strides=[1,1,1,1],padding="SAME")
```

```
    #ReLU ：
    A1 = tf.nn.relu(Z1)
    #Max pool ： 窗口大小：8x8，步伐：8x8，填充方式：“SAME”
    P1 = tf.nn.max_pool(A1,ksize=[1,8,8,1],strides=
[1,8,8,1],padding="SAME")

    #Conv2d ： 步伐：1，填充方式：“SAME”
    Z2 = tf.nn.conv2d(P1,W2,strides=[1,1,1,1],padding="SAME")
    #ReLU ：
    A2 = tf.nn.relu(Z2)
    #Max pool ： 过滤器大小：4x4，步伐：4x4，填充方式：“SAME”
    P2 = tf.nn.max_pool(A2,ksize=[1,4,4,1],strides=
[1,4,4,1],padding="SAME")

    #一维化上一层的输出
    P = tf.contrib.layers.flatten(P2)

    #全连接层（FC）：使用没有非线性激活函数的全连接层
    Z3 = tf.contrib.layers.fully_connected(P,6,activation_fn=None)

    return Z3
```

计算成本

```
def compute_cost(Z3,Y):
    """
    计算成本
    参数：
        Z3 - 正向传播最后一个LINEAR节点的输出，维度为（6，样本数）。
        Y - 标签向量的placeholder，和Z3的维度相同

    返回：
        cost - 计算后的成本

    """

    cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=Z3,labels=Y))

    return cost
```

构建模型

在实现这个模型的时候我们要经历以下步骤：

1. 创建占位符

2. 初始化参数
3. 前向传播
4. 计算成本
5. 反向传播
6. 创建优化器

```
def model(X_train, Y_train, X_test, Y_test, learning_rate=0.009,
          num_epochs=100,minibatch_size=64,print_cost=True,isPlot=True):
    """
    使用TensorFlow实现三层的卷积神经网络
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN ->
FULLYCONNECTED

    参数：
        X_train - 训练数据，维度为(None, 64, 64, 3)
        Y_train - 训练数据对应的标签，维度为(None, n_y = 6)
        X_test - 测试数据，维度为(None, 64, 64, 3)
        Y_test - 训练数据对应的标签，维度为(None, n_y = 6)
        learning_rate - 学习率
        num_epochs - 遍历整个数据集的次数
        minibatch_size - 每个小批量数据块的大小
        print_cost - 是否打印成本值，每遍历100次整个数据集打印一次
        isPlot - 是否绘制图谱

    返回：
        train_accuracy - 实数，训练集的准确度
        test_accuracy - 实数，测试集的准确度
        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)    #确保你的数据和我一样
    seed = 3                 #指定numpy的随机种子
    (m , n_H0, n_W0, n_C0) = X_train.shape
    n_y = Y_train.shape[1]
    costs = []

    #为当前维度创建占位符
    X , Y = create_placeholders(n_H0, n_W0, n_C0, 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，由于框架已经实现了反向传播，我们只需要选择一个优化器就行了
    optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

```
    #全局初始化所有变量
    init = tf.global_variables_initializer()

    #开始运行
    with tf.Session() as sess:
        #初始化参数
        sess.run(init)
        #开始遍历数据集
        for epoch in range(num_epochs):
            minibatch_cost = 0
            num_minibatches = int(m / minibatch_size) #获取数据块的数量
            seed = seed + 1
            minibatches =
cnn_utils.random_mini_batches(X_train,Y_train,minibatch_size,seed)

            #对每个数据块进行处理
            for minibatch in minibatches:
                #选择一个数据块
                (minibatch_X,minibatch_Y) = minibatch
                #最小化这个数据块的成本
                _ , temp_cost = sess.run([optimizer,cost],feed_dict=
{X:minibatch_X, Y:minibatch_Y})

                #累加数据块的成本值
                minibatch_cost += temp_cost / num_minibatches

            #是否打印成本
            if print_cost:
                #每5代打印一次
                if epoch % 5 == 0:
                    print("当前是第 " + str(epoch) + " 代，成本值为：" +
str(minibatch_cost))

            #记录成本
            if epoch % 1 == 0:
                costs.append(minibatch_cost)

        #数据处理完毕，绘制成本曲线
        if isPlot:
            plt.plot(np.squeeze(costs))
            plt.ylabel('cost')
            plt.xlabel('iterations (per tens)')
            plt.title("Learning rate =" + str(learning_rate))
            plt.show()

        #开始预测数据
        ## 计算当前的预测情况
        predict_op = tf.arg_max(Z3,1)
        corrent_prediction = tf.equal(predict_op , tf.arg_max(Y,1))

        ##计算准确度
        accuracy = tf.reduce_mean(tf.cast(corrent_prediction,"float"))
        print("corrent_prediction accuracy= " + str(accuracy))
```

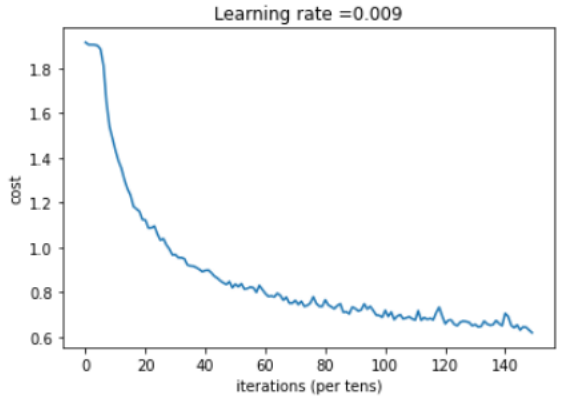
```
        train_accuracy = accuracy.eval({X: X_train, Y: Y_train})
        test_accuary = accuracy.eval({X: X_test, Y: Y_test})

        print("训练集准确度：" + str(train_accuracy))
        print("测试集准确度：" + str(test_accuary))

        return (train_accuracy,test_accuary,parameters)
```

结果

```
no attribute 'index'
当前是第 0 代，成本值为: 1.9145498797297478
当前是第 5 代，成本值为: 1.885212317109108
当前是第 10 代，成本值为: 1.4329352527856827
当前是第 15 代，成本值为: 1.2337106317281723
当前是第 20 代，成本值为: 1.1227323450148106
当前是第 25 代，成本值为: 1.0324298366904259
当前是第 30 代，成本值为: 0.9682332202792168
当前是第 35 代，成本值为: 0.9179412052035332
当前是第 40 代，成本值为: 0.8976681344211102
当前是第 45 代，成本值为: 0.8519235886633396
当前是第 50 代，成本值为: 0.8367729783058167
当前是第 55 代，成本值为: 0.823324590921402
当前是第 60 代，成本值为: 0.79514529556036
当前是第 65 代，成本值为: 0.7861854732036591
当前是第 70 代，成本值为: 0.7643703669309616
当前是第 75 代，成本值为: 0.7520572543144226
当前是第 80 代，成本值为: 0.766705708578229
当前是第 85 代，成本值为: 0.7493216693401337
当前是第 90 代，成本值为: 0.7282395102083683
当前是第 95 代，成本值为: 0.7373781390488148
当前是第 100 代，成本值为: 0.7199424374848604
当前是第 105 代，成本值为: 0.7006734814494848
当前是第 110 代，成本值为: 0.6763854194432497
当前是第 115 代，成本值为: 0.684081107378006
当前是第 120 代，成本值为: 0.6596469841897488
当前是第 125 代，成本值为: 0.6665242183953524
当前是第 130 代，成本值为: 0.6569016017019749
当前是第 135 代，成本值为: 0.6538483817130327
当前是第 140 代，成本值为: 0.7064805589616299
当前是第 145 代，成本值为: 0.6322753597050905
```




```
corrent_prediction accuracy= Tensor("Mean_1:0", shape=(), dtype=float32)
训练集准确度: 0.787037
测试集准确度: 0.76666665
```