暑假学习

7.30(深度学习第一周：建多层神经网络以及应用)

我们来说一下步骤

1. 初始化网络参数
2. 前向传播
    1. 计算一层的中线性求和的部分
    2. 就算激活函数的部分(RELU使用L-1次，Sigmoid使用1次)
    3. 结合线性求和与激活函数
3. 计算误差
4. 反向传播
    1. 线性部分的反向传播公式
    2. 激活函数部分的反向传播公式
    3. 结合线性部分与激活函数的反向传播公式
5. 更新参数

初始化参数

```
def initialize_parameters_deep(layers_dims):
    """
    此函数是为了初始化多层网络参数而使用的函数。
    参数：
        layers_dims - 包含我们网络中每个图层的节点数量的列表

    返回：
        parameters - 包含参数“W1”，“b1”，...，“WL”，“bL”的字典：
                     W1 - 权重矩阵，维度为（layers_dims [1]，layers_dims [1-
1]）
                     bl - 偏向量，维度为（layers_dims [1]，1）
    """
    np.random.seed(3)
    parameters = {}
    L = len(layers_dims)

    for l in range(1,L):
        parameters["W" + str(l)] = np.random.randn(layers_dims[l],
layers_dims[l - 1]) / np.sqrt(layers_dims[l - 1])
        parameters["b" + str(l)] = np.zeros((layers_dims[l], 1))

        #确保我要的数据的格式是正确的
        assert(parameters["W" + str(l)].shape == (layers_dims[l],
layers_dims[l-1]))
        assert(parameters["b" + str(l)].shape == (layers_dims[l], 1))

    return parameters
```

前向传播

Z[l]=W[l]A[l−1]+b

Sigmoid: σ (Z) = σ(WA +b)

ReLU: A =RELU(Z) = max（0,Z)A= RELU(Z) = max(0,Z)

```
def linear_activation_forward(A_prev,W,b,activation):
    """
    实现LINEAR-> ACTIVATION 这一层的前向传播

    参数：
        A_prev - 来自上一层（或输入层）的激活，维度为(上一层的节点数量，示例数）
        W - 权重矩阵，numpy数组，维度为（当前层的节点数量，前一层的大小）
        b - 偏向量，numpy阵列，维度为（当前层的节点数量，1）
        activation - 选择在此层中使用的激活函数名，字符串类型，【"sigmoid" |
"relu"】

    返回：
        A - 激活函数的输出，也称为激活后的值
        cache - 一个包含“linear_cache”和“activation_cache”的字典，我们需要存储它
以有效地计算后向传递
    """

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert(A.shape == (W.shape[0],A_prev.shape[1]))
    cache = (linear_cache,activation_cache)

    return A,cache
```

```
def L_model_forward(X,parameters):
    """
    实现[LINEAR-> RELU] *（L-1） - > LINEAR-> SIGMOID计算前向传播，也就是多层网络
的前向传播，为后面每一层都执行LINEAR和ACTIVATION

    参数：
        X - 数据，numpy数组，维度为（输入节点数量，示例数）
        parameters - initialize_parameters_deep（）的输出

    返回：
        AL - 最后的激活值
        caches - 包含以下内容的缓存列表：
                 linear_relu_forward（）的每个cache（有L-1个，索引为从0到L-2）
                 linear_sigmoid_forward（）的cache（只有一个，索引为L-1）
```

```
    """
    caches = []
    A = X
    L = len(parameters) // 2
    for l in range(1,L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' +
str(l)], parameters['b' + str(l)], "relu")
        caches.append(cache)

    AL, cache = linear_activation_forward(A, parameters['W' + str(L)],
parameters['b' + str(L)], "sigmoid")
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL,caches
```

计算成本

```
def compute_cost(AL,Y):
    """
    实施等式（4）定义的成本函数。

    参数：
        AL - 与标签预测相对应的概率向量，维度为（1，示例数量）
        Y - 标签向量（例如：如果不是猫，则为0，如果是猫则为1），维度为（1，数量）

    返回：
        cost - 交叉熵成本
    """
    m = Y.shape[1]
    cost = -np.sum(np.multiply(np.log(AL),Y) + np.multiply(np.log(1 - AL),
1 - Y)) / m

    cost = np.squeeze(cost)
    assert(cost.shape == ())

    return cost
```

反向传播 dW[l]=∂W[l]∂L=1/m*dZ^[l]A^[l−1]T

db[l]=∂L/∂b[l]=1/m*1∑mdZl

dA[l−1]=∂L/∂A[l−1]=W^[l]TdZ[l]

线性部分(第L层)

```
def linear_backward(dZ,cache):
    """
```

```
    为单层实现反向传播的线性部分（第L层）

    参数：
         dZ - 相对于（当前第l层的）线性输出的成本梯度
         cache - 来自当前层前向传播的值的元组（A_prev，W，b）

    返回：
         dA_prev - 相对于激活（前一层l-1）的成本梯度，与A_prev维度相同
         dW - 相对于W（当前层l）的成本梯度，与W的维度相同
         db - 相对于b（当前层l）的成本梯度，与b维度相同
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis=1, keepdims=True) / m
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

sigmoid_backward:实现了sigmoid（）函数的反向传播，你可以这样调用它：

dZ = sigmoid_backward(dA, activation_cache)

relu_backward: 实现了relu（）函数的反向传播，你可以这样调用它：

dZ = relu_backward(dA, activation_cache)

线性激活部分

```
def linear_activation_backward(dA,cache,activation="relu"):
    """
    实现LINEAR-> ACTIVATION层的后向传播。

    参数：
         dA - 当前层l的激活后的梯度值
         cache - 我们存储的用于有效计算反向传播的值的元组（值为linear_cache，
activation_cache）
         activation - 要在此层中使用的激活函数名，字符串类型，【"sigmoid" |
"relu"】
    返回：
         dA_prev - 相对于激活（前一层l-1）的成本梯度值，与A_prev维度相同
         dW - 相对于W（当前层l）的成本梯度值，与W的维度相同
         db - 相对于b（当前层l）的成本梯度值，与b的维度相同
    """
    linear_cache, activation_cache = cache
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
```

```
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev,dW,db
```

多层模型向后传播

```
def L_model_backward(AL,Y,caches):
    """
    对[LINEAR-> RELU] *（L-1） - > LINEAR - > SIGMOID组执行反向传播，就是多层网络的向后传播

    参数：
     AL - 概率向量，正向传播的输出（L_model_forward（））
     Y - 标签向量（例如：如果不是猫，则为0，如果是猫则为1），维度为（1，数量）
     caches - 包含以下内容的cache列表：
                linear_activation_forward（"relu"）的cache，不包含输出层
                linear_activation_forward（"sigmoid"）的cache

    返回：
     grads - 具有梯度值的字典
              grads [“dA”+ str（l）] = ...
              grads [“dW”+ str（l）] = ...
              grads [“db”+ str（l）] = ...
    """
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache, "sigmoid")

    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache, "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

更新参数

```
def update_parameters(parameters, grads, learning_rate):
    """
    使用梯度下降更新参数

    参数：
     parameters - 包含你的参数的字典
     grads - 包含梯度值的字典，是L_model_backward的输出

    返回：
     parameters - 包含更新参数的字典
                  参数[“W”+ str（l）] = ...
                  参数[“b”+ str（l）] = ...
    """
    L = len(parameters) // 2 #整除
    for l in range(L):
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] -
learning_rate * grads["dW" + str(l + 1)]
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] -
learning_rate * grads["db" + str(l + 1)]

    return parameters
```

多层神经网络

```
def L_layer_model(X, Y, layers_dims, learning_rate=0.0075,
num_iterations=3000, print_cost=False,isPlot=True):
    """
    实现一个L层神经网络：[LINEAR-> RELU] *（L-1） - > LINEAR-> SIGMOID。

    参数：
        X - 输入的数据，维度为(n_x，例子数)
        Y - 标签，向量，0为非猫，1为猫，维度为(1,数量)
        layers_dims - 层数的向量，维度为(n_y,n_h,···,n_h,n_y)
        learning_rate - 学习率
        num_iterations - 迭代的次数
        print_cost - 是否打印成本值，每100次打印一次
        isPlot - 是否绘制出误差值的图谱

    返回：
     parameters - 模型学习的参数。 然后他们可以用来预测。
    """
    np.random.seed(1)
    costs = []

    parameters = initialize_parameters_deep(layers_dims)

    for i in range(0,num_iterations):
        AL , caches = L_model_forward(X,parameters)

        cost = compute_cost(AL,Y)
```

```
        grads = L_model_backward(AL,Y,caches)

        parameters = update_parameters(parameters,grads,learning_rate)

        #打印成本值，如果print_cost=False则忽略
        if i % 100 == 0:
            #记录成本
            costs.append(cost)
            #是否打印成本值
            if print_cost:
                print("第", i ,"次迭代，成本值为：" ,np.squeeze(cost))
    #迭代完成，根据条件绘制图
    if isPlot:
        plt.plot(np.squeeze(costs))
        plt.ylabel('cost')
        plt.xlabel('iterations (per tens)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()
    return parameters
```

训练结果

```
第 0 次迭代，成本值为： 0.715731513413713
第 100 次迭代，成本值为： 0.6747377593469114
第 200 次迭代，成本值为： 0.6603365433622126
第 300 次迭代，成本值为： 0.6462887802148751
第 400 次迭代，成本值为： 0.6298131216927774
第 500 次迭代，成本值为： 0.606005622926534
第 600 次迭代，成本值为： 0.5690041263975134
第 700 次迭代，成本值为： 0.519796535043806
第 800 次迭代，成本值为： 0.464157167862823
第 900 次迭代，成本值为： 0.4084203004829893
第 1000 次迭代，成本值为： 0.37315499216069026
第 1100 次迭代，成本值为： 0.3057237457304711
第 1200 次迭代，成本值为： 0.26810152847740837
第 1300 次迭代，成本值为： 0.23872474827672668
第 1400 次迭代，成本值为： 0.20632263257914704
第 1500 次迭代，成本值为： 0.17943886927493605
第 1600 次迭代，成本值为： 0.1579873581880166
第 1700 次迭代，成本值为： 0.14240413012274525
第 1800 次迭代，成本值为： 0.12865165997888856
第 1900 次迭代，成本值为： 0.11244314998164931
第 2000 次迭代，成本值为： 0.08505631034983409
第 2100 次迭代，成本值为： 0.05758391198617392
第 2200 次迭代，成本值为： 0.044567534546994664
第 2300 次迭代，成本值为： 0.038082751666004236
第 2400 次迭代，成本值为： 0.03441074901842096
```

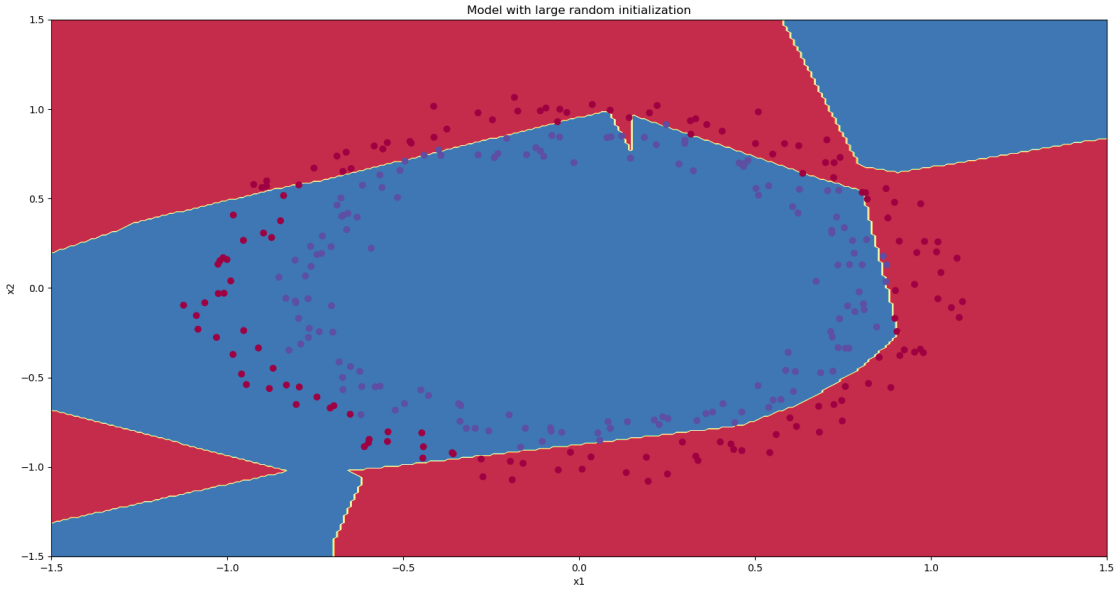
```
准确度为: 0.9952153110047847
准确度为: 0.78
```

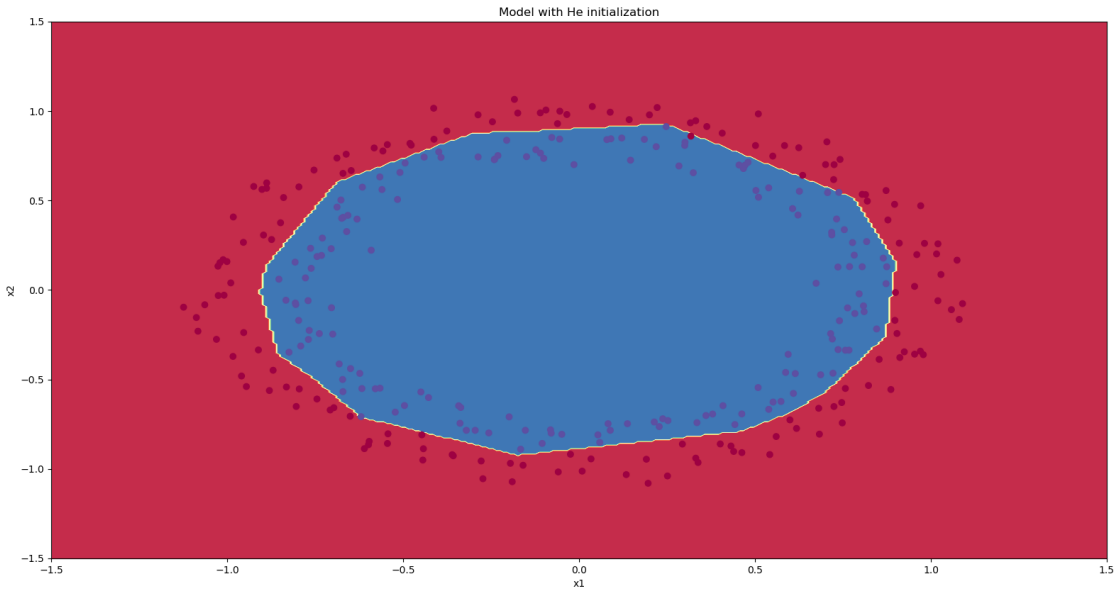
8.1(深度学习中的抑梯度异常初始化，正则化和梯度检测)

抑梯度异常初始化

```
    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layers_dims[l],
layers_dims[l - 1]) * np.sqrt(2 / layers_dims[l - 1])
        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
```

这就是异梯度异常初始化，我们可以看看这个和随机初始化的效果差别。





初始化的模型将蓝色和红色的点在少量的迭代中很好地分离出来，总结一下：

1. 不同的初始化方法可能导致性能最终不同
2. 随机初始化有助于打破对称，使得不同隐藏层的单元可以学习到不同的参数。
3. 初始化时，初始值不宜过大。
4. He初始化搭配ReLU激活函数常常可以得到不错的效果。

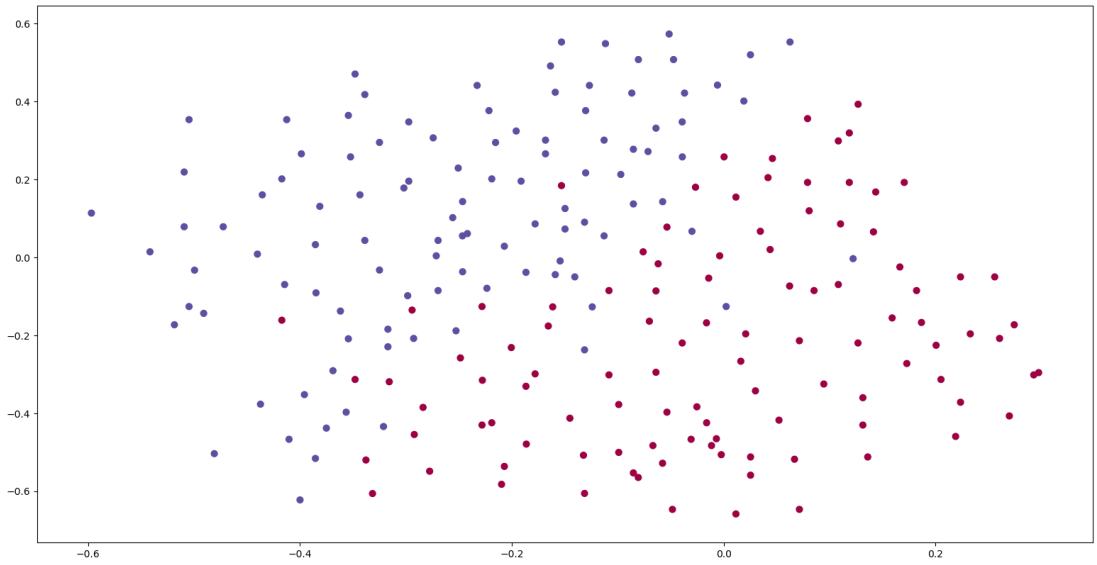
正则化

在深度学习中，如果数据集没有足够大的话，可能会导致一些过拟合的问题。过拟合导致的结果就是在训练集上有着很高的精确度，但是在遇到新的样本时，精确度下降会很严重。为了避免过拟合的问题，接下来我们要讲解的方式就是正则化。

我们要做以下三件事，来对比出不同的模型的优劣：

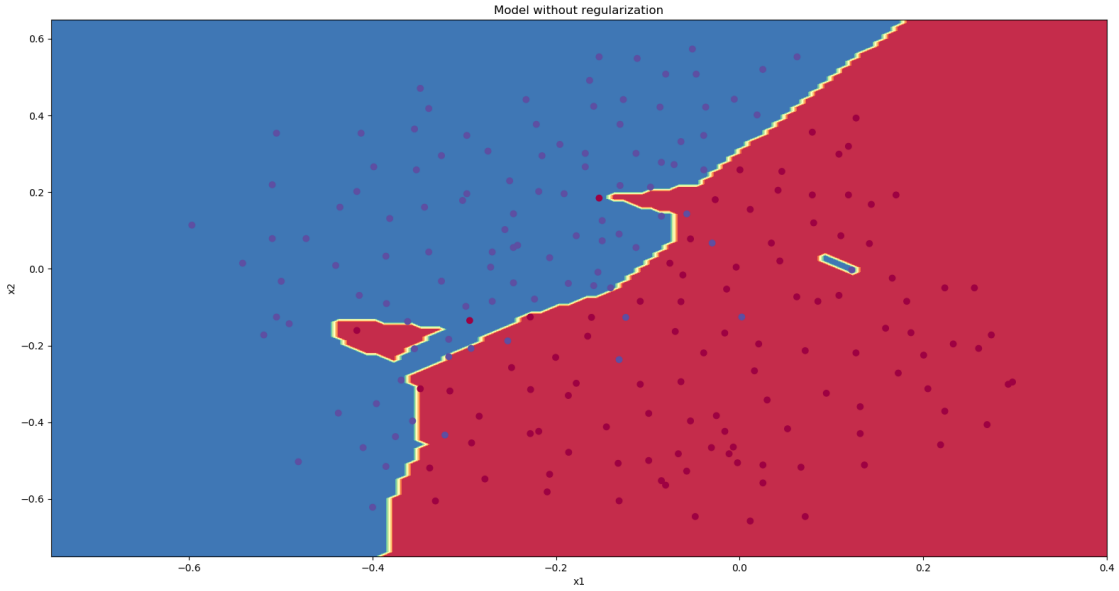
1. 不使用正则化
2. 使用正则化
    1. 用L2正则化
    2. 使用随机节点删除

数据集如下



不使用正则化

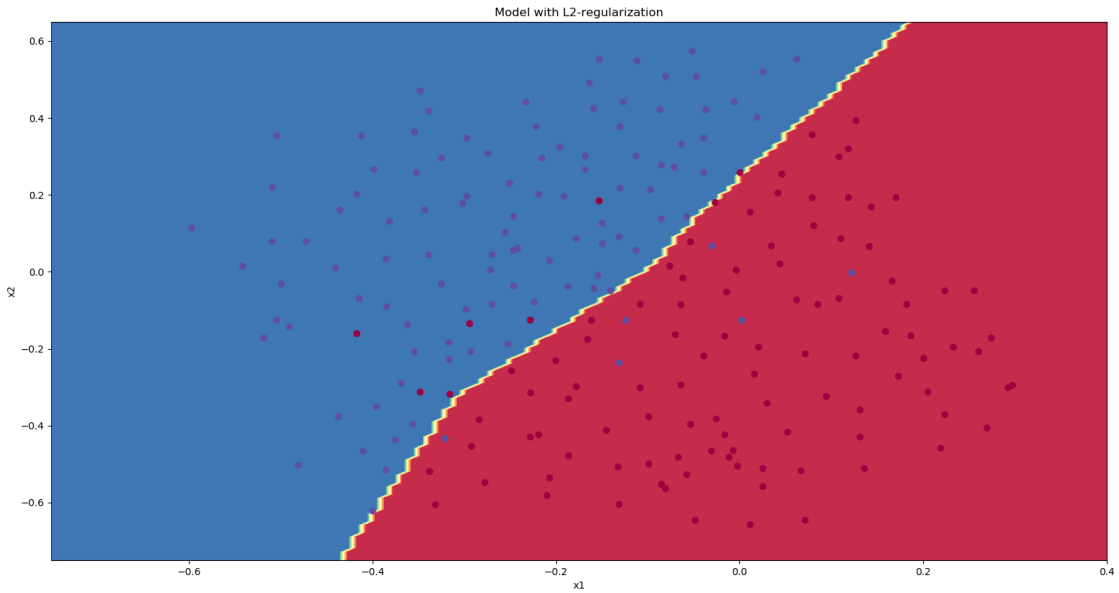
而不使用正则化的结果如下



我们可以看到很明显的过拟合，我们可以使用正则化来使拟合下降，即平滑的分界线

L2正则化

L2正则化即在成本函数加上 1/m * λ/2 * np.sum(np.square(Wl)) 相应的向后传播时，dw也要加上((λ* W2) / m)



我们可以很明显的看见过拟合消失，边界较为平滑。

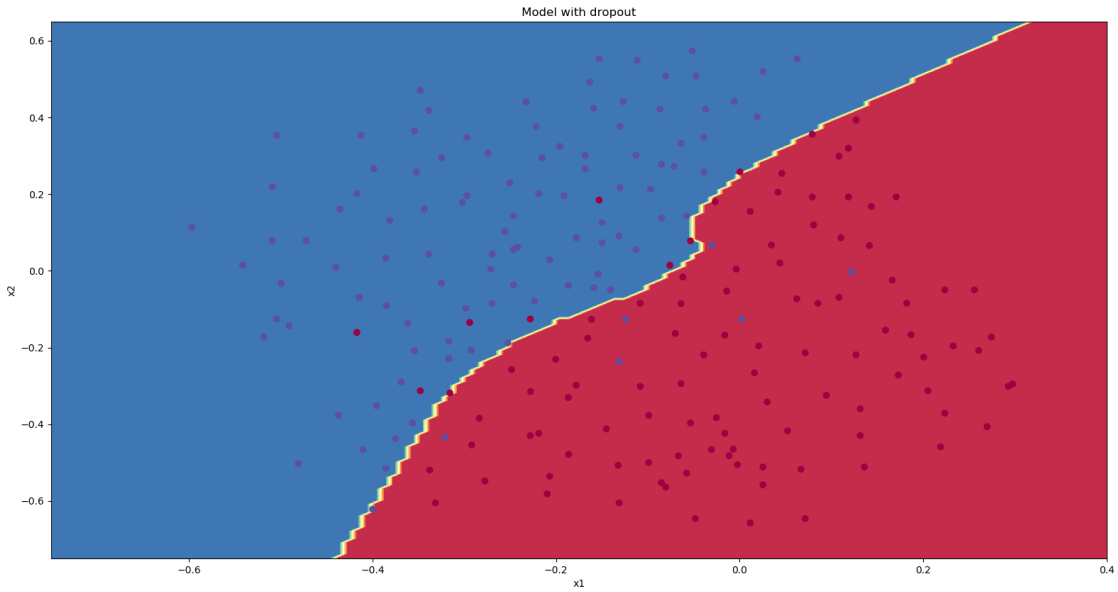
λ的值是可以使用开发集调整时的超参数。L2正则化会使决策边界更加平滑。如果λ太大，也可能会“过度平滑”，从而导致模型高偏差。L2正则化实际上在做什么？L2正则化依赖于较小权重的模型比具有较大权重的模型更简单这样的假设，因此，通过削减成本函数中权重的平方值，可以将所有权重值逐渐改变到到较小的值。权值数值高的话会有更平滑的模型，其中输入变化时输出变化更慢，但是你需要花费更多的时间。L2正则化对以下内容有影响：

1. 成本计算 ： 正则化的计算需要添加到成本函数中
2. 反向传播功能 ：在权重矩阵方面，梯度计算时也要依据正则化来做出相应的计算
3. 重量变小（“重量衰减”） ：权重被逐渐改变到较小的值。

使用随机删除

我们使用Dropout来进行正则化，Dropout的原理就是每次迭代过程中随机将其中的一些节点失效。当我们关闭一些节点时，我们实际上修改了我们的模型。背后的想法是，在每次迭代时，我们都会训练一个只使用一部分神经元的不同模型。随着迭代次数的增加，我们的模型的节点会对其他特定节点的激活变得不那么敏感，因为其他节点可能在任何时候会失效。 下面我们将关闭第一层和第三层的一些节点，我们需要做以下四步：

1. 在视频中，吴恩达老师讲解了使用np.random.rand() 来初始化和 D[1]。D[1]和A[1]有同样的大小
2. 如果 D [1] 低于 (keep_prob)的值我们就把它设置为0，如果高于(keep_prob)的值我们就设置为1。
3. 把A [1] 更新为 A [1] ＊ D [1] (我们已经关闭了一些节点)。我们可以使用 D 作为掩码。我们做矩阵相乘的时候，关闭的那些节点（值为0）就会不参与计算，因为0乘以任何值都为0。
4. 使用 A [1] 除以 keep_prob。这样做的话我们通过缩放就在计算成本的时候仍然具有相同的期望值，这叫做反向dropout。



我们可以看到，正则化会把训练集的准确度降低，但是测试集的准确度提高了，所以，我们这个还是成功了。

梯度校验

在我们执行反向传播的计算过程中，反向传播函数的计算过程是比较复杂的。为了验证我们得到的反向传播函数是否正确，现在你需要编写一些代码来验证反向传播函数的正确性。即验证dw和db的值是否正确，是否出现了bug。

我们先将通过将所有参数（W1，b1，W2，b2，W3，b3）整形为向量并将它们连接起来而获得。得到一个向量parameters。

1. 计算 J_plus[i]:
    1. θ[i+]设置为 θ[i+] + ε
    2. 然后根据forward_interation()来计算 J_puls[i]

2. 计算 J_plus[i]同上
3. 计算gradapprox[i]
4. 计算误差

一般来说，当difference小于 10^(-7)，就认为我们的计算是正确的。