计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目： Homework 2_2		学号：201900130024
日期：2021.10.20	班级： 数据 19	姓名： 刘士渤

Email：liuburger@qq.com

实验目的：

完成 Regularization 和 Batch Normalization

实验软件和硬件环境：

VScode JupyterNoteBook

联想拯救者 Y7000p

实验原理和方法：

neural network

实验步骤：（不要求罗列完整源代码）

1. 补全 Regularization.ipynb

   根据提示：

Exercise: Implement `compute_cost_with_regularization()` which computes the cost given by formula (2). To calculate $\sum\limits_k\sum\limits_j W_{k,j}^{[l]2}$, use :

`np.sum(np.square(W1))`

得代码：

```
### START CODE HERE ### (approx. 1 line)
L2_regularization_cost = lambd*(np.sum(np.square(W1))+np.sum(np.square(W2))+np.sum(np.square(W3)))/(2*m)
### END CODER HERE ###
```

根据提示：

Exercise: Implement the changes needed in backward propagation to take into account regularization. The changes only concern dW1, dW2 and dW3. For each, you have to add the regularization term's gradient ($\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$).

得代码：

```
### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd/m)*W3
### END CODE HERE ###
```

```
### START CODE HERE ### (approx. 1 line)
dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd/m)*W2
### END CODE HERE ###
```

```
### START CODE HERE ### (approx. 1 line)
dW1 = 1./m * np.dot(dZ1, X.T) + (lambd/m)*W1
### END CODE HERE ###
```

根据提示：

1. In lecture, we dicussed creating a variable $d^{[1]}$ with the same shape as $a^{[1]}$ using `np.random.rand()` to randomly get numbers between 0 and 1. Here, you will use a vectorized implementation, so create a random matrix $D^{[1]} = [d^{[1](1)} d^{[1](2)} \ldots d^{[1](m)}]$ of the same dimension as $A^{[1]}$.
2. Set each entry of $D^{[1]}$ to be 0 with probability (`1-keep_prob`) or 1 with probability (`keep_prob`), by thresholding values in $D^{[1]}$ appropriately. Hint: to set all the entries of a matrix X to 0 (if entry is less than 0.5) or 1 (if entry is more than 0.5) you would do: `X = (X < 0.5)`. Note that 0 and 1 are respectively equivalent to False and True.
3. Set $A^{[1]}$ to $A^{[1]} * D^{[1]}$. (You are shutting down some neurons). You can think of $D^{[1]}$ as a mask, so that when it is multiplied with another matrix, it shuts down some of the values.
4. Divide $A^{[1]}$ by `keep_prob`. By doing this you are assuring that the result of the cost will still have the same expected value as without drop-out. (This technique is also called inverted dropout.)

得代码：

```
### START CODE HERE ### (approx. 4 lines)
D1 = np.random.rand(A1.shape[0],A1.shape[1])
D1 = (D1<keep_prob)
A1 *= D1
A1 /= keep_prob
### END CODE HERE ###
```

```
### START CODE HERE ### (approx. 4 lines)
D2 = np.random.rand(A2.shape[0],A2.shape[1])
D2 = (D2<keep_prob)
A2 *= D2
A2 /= keep_prob
### END CODE HERE ###
```

根据提示：

1. You had previously shut down some neurons during forward propagation, by applying a mask $D^{[1]}$ to `A1`. In backpropagation, you will have to shut down the same neurons, by reapplying the same mask $D^{[1]}$ to `dA1`.
2. During forward propagation, you had divided `A1` by `keep_prob`. In backpropagation, you'll therefore have to divide `dA1` by `keep_prob` again (the calculus interpretation is that if $A^{[1]}$ is scaled by `keep_prob`, then its derivative $dA^{[1]}$ is also scaled by the same `keep_prob`).

得代码：

```
### START CODE HERE ###
dA2 *= D2
dA2 /= keep_prob
### END CODE HERE ###
```

```
### START CODE HERE ###
dA1 *= D1
dA1 /= keep_prob
### END CODE HERE ###
```

2. 补全 layers.py

batchnorm_forward:

由课件中的均值、方差、正则化样本和输出的数学式

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j}-\mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j}-\mu_j}{\sqrt{\sigma_j^2+\varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

得代码:

train（用到动量的方法）:

```
sample_mean=np.mean(x,axis=0)
sample_var=np.var(x,axis=0)
x_hat=(x-sample_mean)/np.sqrt(sample_var+eps)
out=gamma*x_hat+beta
cache=(x,gamma,beta,x_hat,sample_mean,sample_var,eps)

running_mean=momentum*running_mean+(1-momentum)*sample_mean
running_var=momentum*running_var+(1-momentum)*sample_var
```

test:

```
x_hat=(x-running_mean)/np.sqrt(running_var+eps)
out=gamma*x_hat+beta
```

batchnorm_backward：

运用 *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* 一文提到的反向链式法则：

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^m -2(x_i-\mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i-\mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

得代码（真没想到σ在代码中还能被识别）：

```
x,gamma,beta,x_hat,sample_mean,sample_var,eps=cache
N=x.shape[0]
dx_hat=dout*gamma
dσ2=np.sum((x-sample_mean)*dx_hat,axis=0)*(-0.5*((sample_var+eps)**-1.5))
dmiu=np.sum(-(sample_var+eps)**-0.5*dx_hat,axis=0)+dσ2*(-2*np.sum(x-sample_mean,axis=0))/N
dx=dx_hat*(sample_var+eps)**-0.5+dσ2*2*(x-sample_mean)/N+dmiu/N
dgamma=np.sum(dout*x_hat,axis=0)
dbeta=np.sum(dout,axis=0)
```

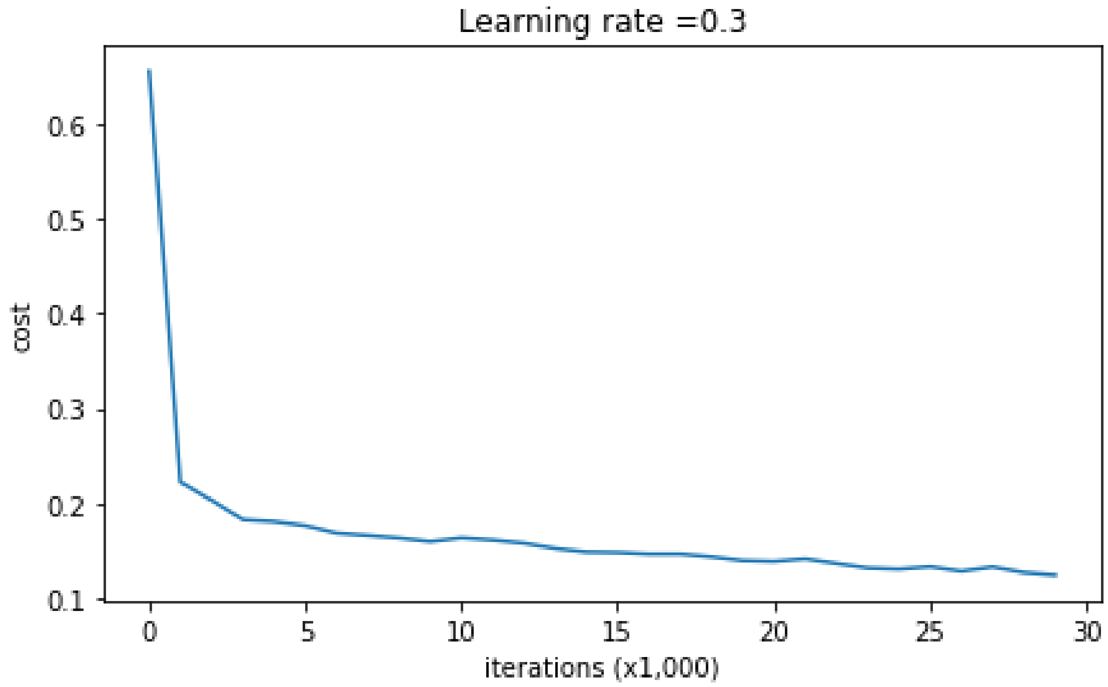
batchnorm_backward_alt（类似 batchnorm_backward）：

```
x,gamma,beta,x_hat,sample_mean,sample_var,eps=cache
m=dout.shape[0]
dxhat=dout*gamma
dvar=(dxhat*(x-sample_mean)*(-0.5)*np.power(sample_var+eps,-1.5)).sum(axis=0)
dmean=np.sum(dxhat*(-np.power(sample_var+eps,-0.5)),axis=0)
dmean+=dvar*np.sum(-2*(x-sample_mean),axis=0)/m
dx=dxhat*np.power(sample_var+eps,-0.5)+dvar*2*(x-sample_mean)/m+dmean/m
dgamma=np.sum(dout*x_hat,axis=0)
dbeta=np.sum(dout,axis=0)
```

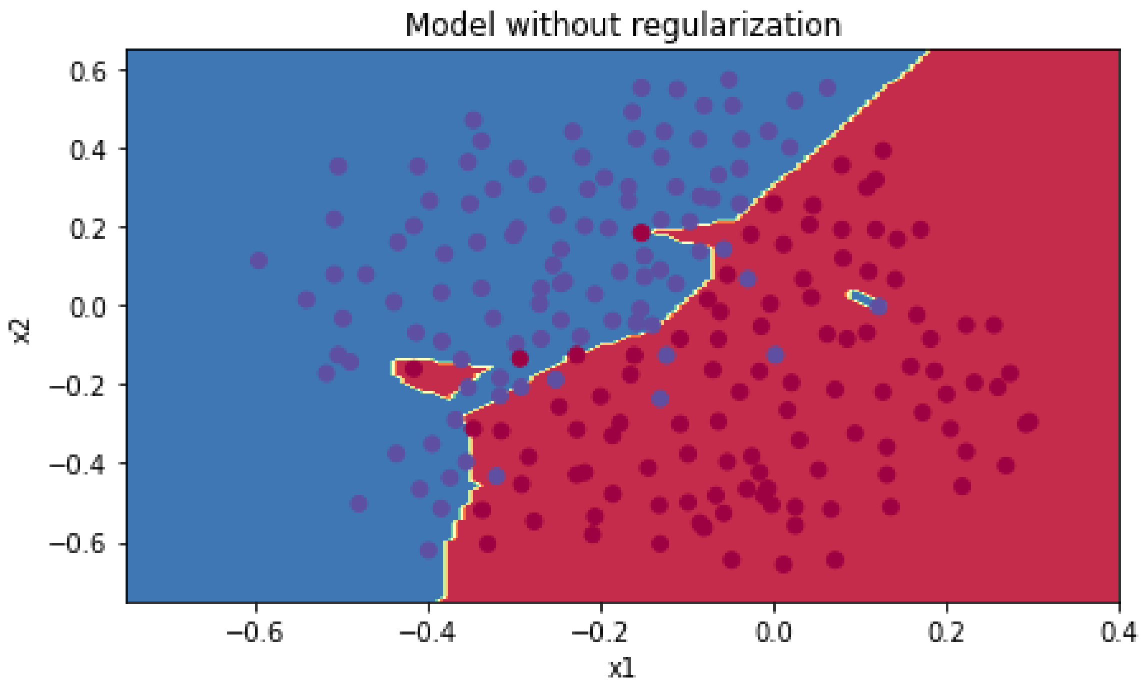
结论分析与体会：

1. Regularization：

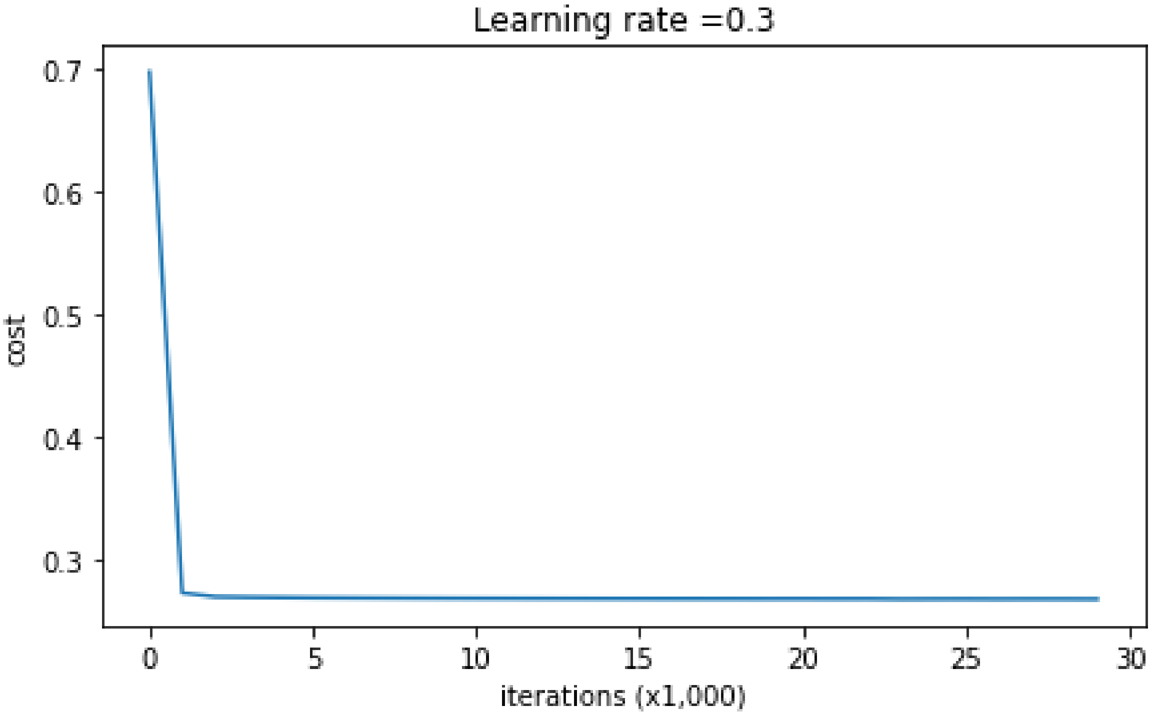
没有正则项的话，cost 在迭代 100 次之前之后下降较慢



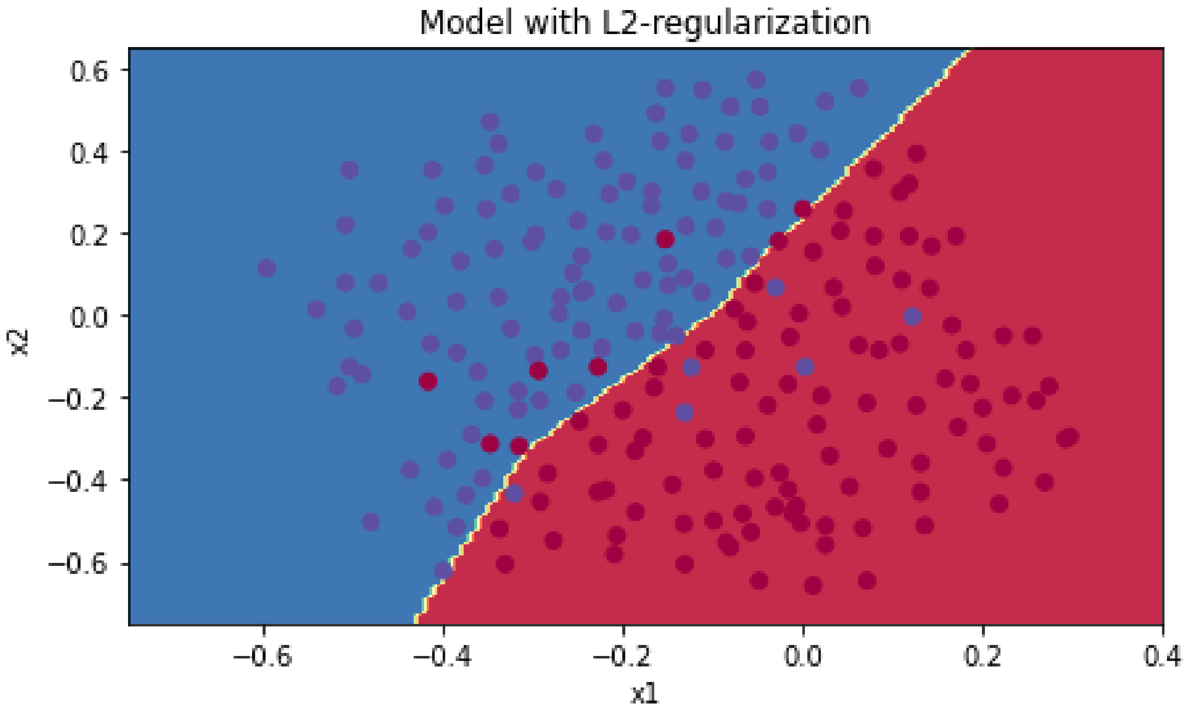
因为没有正则项，所以出现了过拟合，大的红蓝区域都有小的蓝红区域；测试准确率为 91.5%。



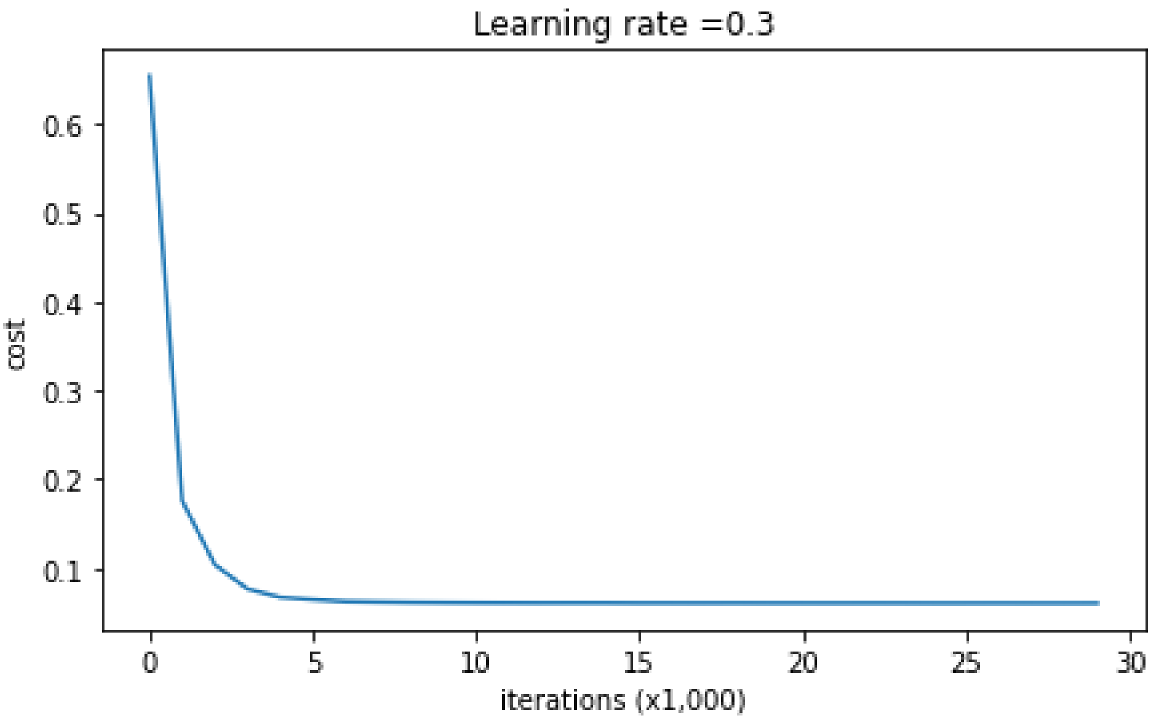
加入 L2 正则项，cost 很早就收敛了，但还是大于 0.2 的，这一点不如没有正则项：



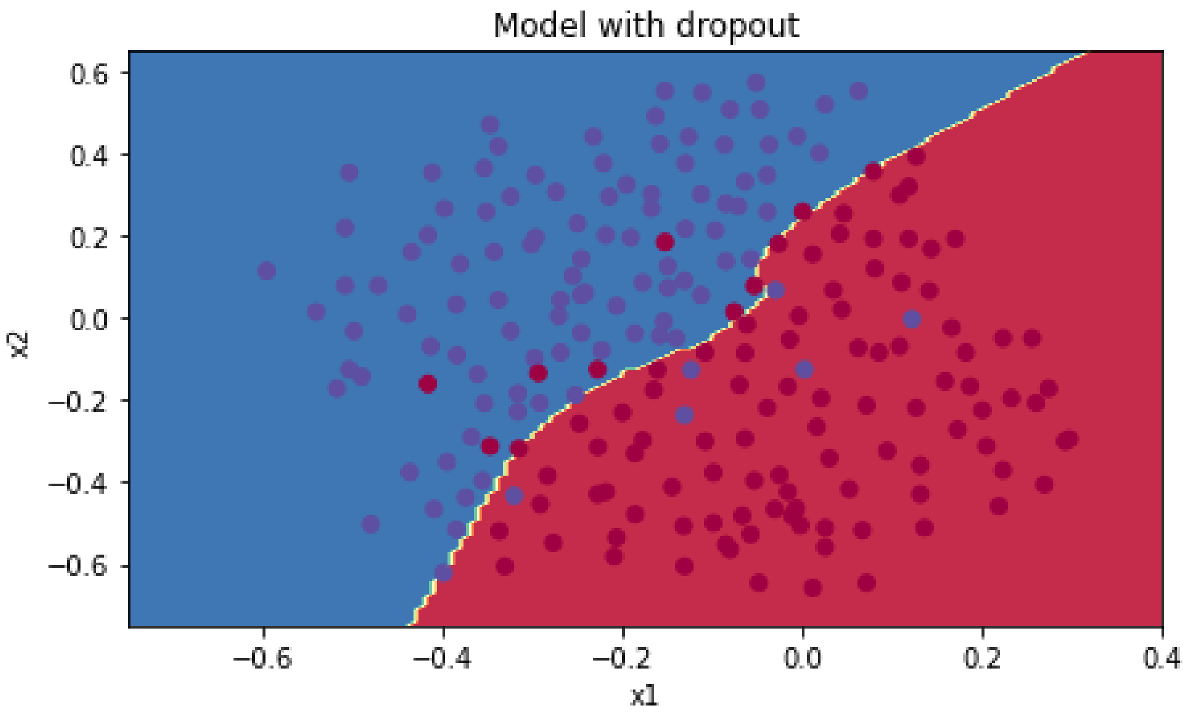
相较于没有正则项，过拟合的现象不存在了，测试准确率为 93%，提高了 1.5%。



采用 Dropout，cost 既收敛得早，又降到了 0.1 以下：



同样的没有过拟合，相较于 L2 正则项，边界不那么平滑，测试准确率为 95%，提高了 2%。



2. BatchNormalization：

forward：

相较于 BatchNormalization 之前，均值和方差都变得很小了；

不同的 gamma 和 beta 组合会使均值和方差的大小产生变化。

（训练集）

```
Before batch normalization:
  means:  [ -2.3814598   -13.18038246   1.91780462]
  stds:   [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means:  [4.66293670e-17 5.27355937e-17 9.57567359e-18]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means:  [11. 12. 13.]
  stds:   [0.99999999 1.99999999 2.99999999]
```

（测试集）

```
After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:   [1.01531428 1.01238373 0.97819988]
```

backward（误差都很小，方法可行）：

```
dx error:  1.7029261167605239e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

alternative backward（每次运行的 speedup 都不太一样，但都在 1 以上）：

```
dx difference:  1.07333843309325e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.05x
```

就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1—3 道问答题：

1. 一开始误以为下式中的 L2 正则项需要 sum 套 sum，因为有 3 个∑

$$J_{regularized} = \underbrace{-\frac{1}{m} \sum\limits_{i = 1}^{m} \large{(}\small y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1- a^{[L](i)}\right) \large{)} }_\text{cross-entropy cost} + \underbrace{\frac{1}{m} \frac{\lambda}{2} \sum\limits_l\sum\limits_k\sum\limits_j W_{k,j}^{[l]2} }_\text{L2 regularization cost}$$

   但随后明白 np.sum(np.square(W))已经解决了

$$\sum\limits_k\sum\limits_j W_{k,j}^{[l]2}$$

   所以只需要把三项加起来。

2. keep_prob 是一个神经元保持活动的概率，

```
D = (D<keep_prob)
```

   是把概率小于 keep_prob 的变成 1，把概率大于 keep_prob 的变成 0，感觉就是应该保持的没保持，不应该保持的保持了。但是这样写运行结果是正确的，

```
D = (D>keep_prob)
```

   反而不对了，不知道为什么。