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

Experiment 4

实验题目：这次作业中，我们需要掌握基本的神经网络调整技巧，并尝试改进深度神经网络:使用超参数调整、正则化和优化、BatchNormalization

实验目标：这次我们需要实现两个子任务:正则化和BatchNormalization

实验学生基本信息：

- 姓名：陈佳睿
- 学号：201700301042
- 班级：17人工智能班

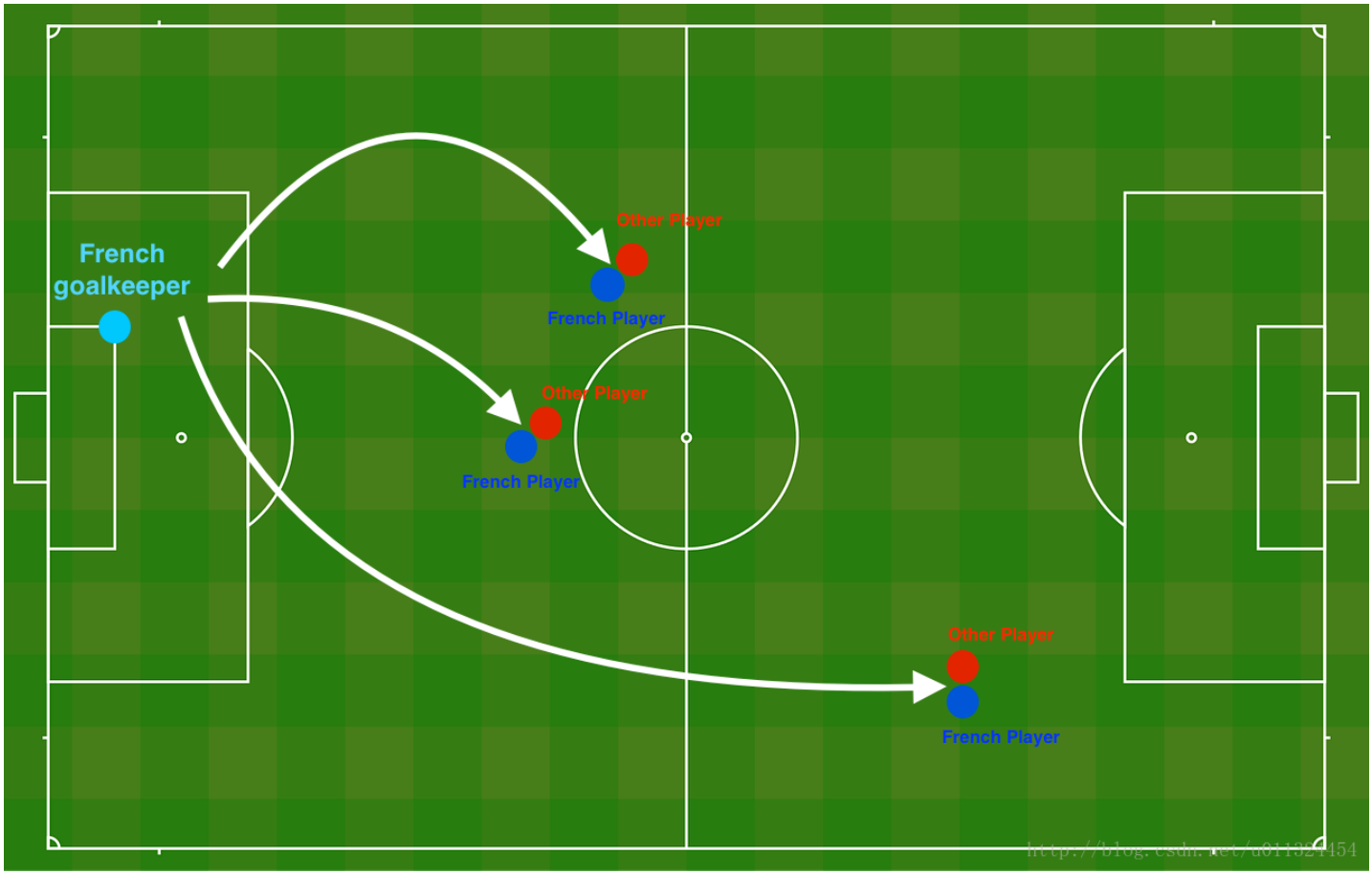
日期：2020.3.20

实验目的：利用正则化和BatchNormalization方法来提高神经网络的表现

一.正则化 Regularization

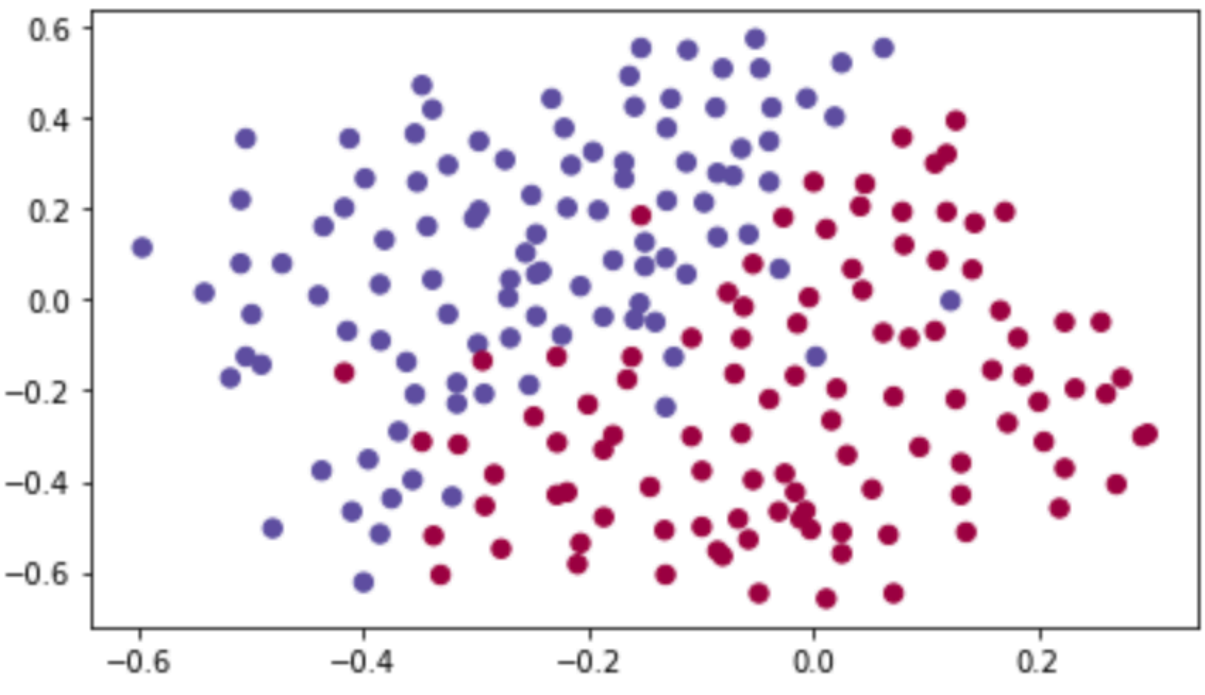
深度学习模型有很大的灵活性和容量，如果训练数据集不够大，过度拟合可能会成为一个严重的问题。当然，它在训练集上做得很好，但是学习网络不能推广到它从未见过的新例子上!
我学习的目标： Use regularization in your deep learning models

正则化实验目的：使用AI算法为法国足球公司推荐法国守门员应该踢球的位置



首先使用 `load_2D_dataset()` 函数加载数据

```
train_X, train_Y, test_X, test_Y = load_2D_dataset()
```



每个点对应的是足球场上的一个位置，在法国守门员从足球场地左侧射门后，足球运动员用

头部击球。

- 圆点是蓝色的，表示法国队队员可以成功用头击球
- 圆点是红色的，表示对方球员可以用头部击球

目标:使用一个深度学习模型来找到守门员应该踢球的位置。

数据集的分析:这个数据集有点小噪音，但它看起来像一条对角线，将左上角(蓝色)和右下角(红色)分隔开，效果很好。

我将首先尝试一个非正则化模型。然后我将学习如何规范它，并决定我将选择哪种模式来解决法国足球公司的问题。

```
1-Non-regularized model
```

在正则化模式下，将输入变量 `lambd` (正则化超参数,标量）设置为非零值
在Dropout模式下，将变量 `keep_prob` (神经元在dropout保持活动的概率，标量)设置为小于1的值

首先尝试不需要任何正则化的模型

L2 正则化函数: `compute_cost_with_regularization()` and `backward_propagation_with_regularization()`

Dropout函数: `forward_propagation_with_dropout()` and `backward_propagation_with_dropout()`

三层神经网络的结构：
`LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID`

可以将 `lambd` 和 `keep_prob` 理解为我们设置的一个信号量来控制我们是否使用正则和dropout，当然是否dropout不同于正则，它是一个概率问题
相关代码注解：

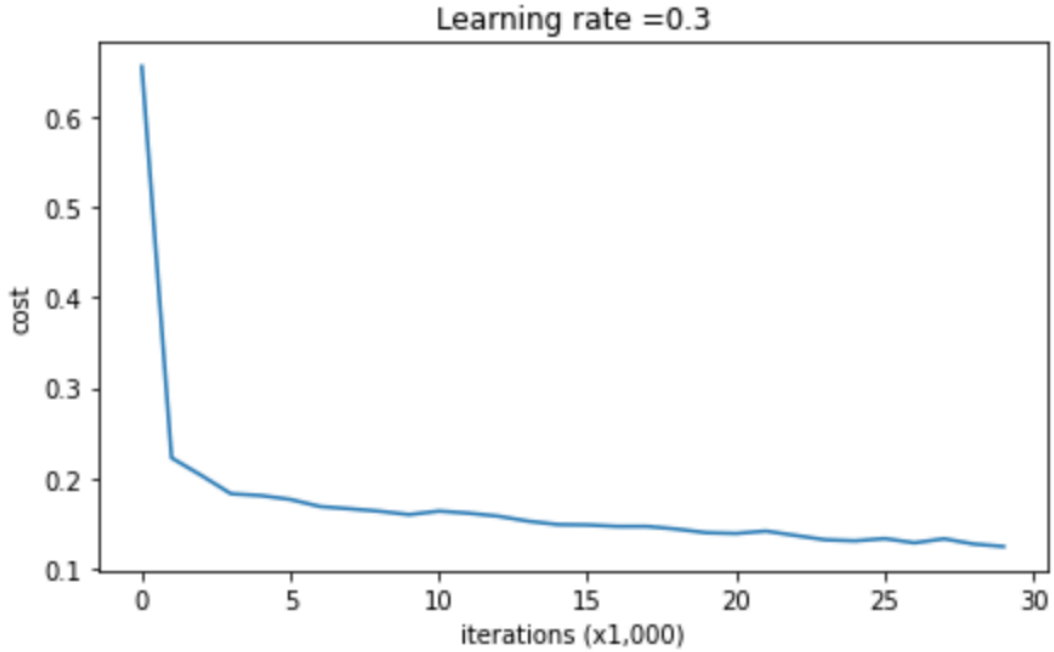
```
# Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
if keep_prob == 1:
    a3, cache = forward_propagation(X, parameters)
elif keep_prob < 1:
    a3, cache = forward_propagation_with_dropout(X, parameters, keep_prob)

# Cost function
if lambd == 0:
    cost = compute_cost(a3, Y)
else:
    cost = compute_cost_with_regularization(a3, Y, parameters, lambd)

# Backward propagation.
assert(lambd==0 or keep_prob==1)    # it is possible to use both L2 regularization and dropout,
                                    # but this assignment will only explore one at a time
if lambd == 0 and keep_prob == 1:
    grads = backward_propagation(X, Y, cache)
elif lambd != 0:
    grads = backward_propagation_with_regularization(X, Y, cache, lambd)
elif keep_prob < 1:
    grads = backward_propagation_with_dropout(X, Y, cache, keep_prob)
```

我们不使用正则，在数据集的训练集和测试集上得到的结果曲线：

```
Cost after iteration 0: 0.6557412523481002
Cost after iteration 10000: 0.1632998752572417
Cost after iteration 20000: 0.13851642423284755
```

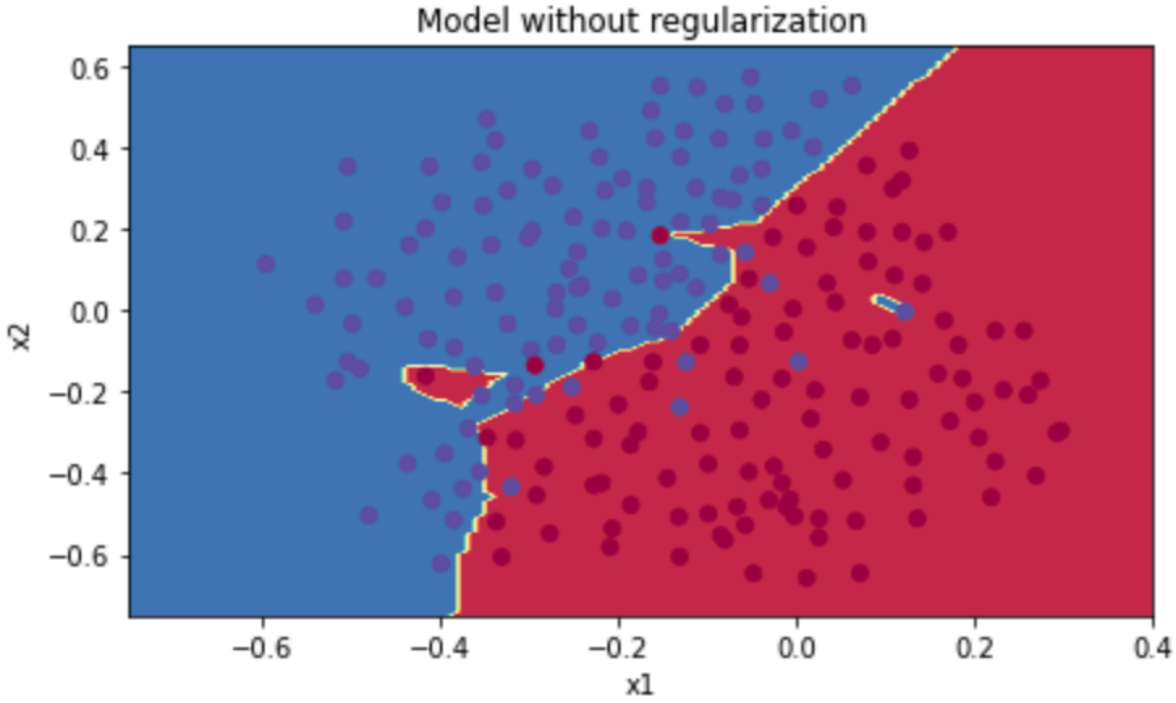



```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

训练集准确率:94.8%
测试集准确率:91.5%

但是这只是我们模型的baseline，接下来我们来观察正则对于模型准确率的影响

非正则化模型显然过拟合了，它是对噪声点的拟合
可视化结果后，我们更加容易看出来



2 - L2 Regularization

第一种防止过拟合的标准方法是使用L2正则
我们需要适当地修改损失函数，从

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

修改到：

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

我需要实现 `compute_cost_with_regularization()` 函数，当我们需要计算$\sum_{k}\sum_{j}W_{k,j}^{[l]2}$时，我们使用 `np.sum(np.square(Wl))`

我们对于$W^{[1]}$, $W^{[2]}$ and $W^{[3]}$的和乘以$\frac{1}{m}\frac{\lambda}{2}$来得到 L2正则损失

因为我们对于损失函数加入了正则项，所以对于后向传播的计算我们也需要进行修改。
进行的修改只与dW1, dW2和dW3有关，加入正则项梯度
$\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$

使用测试样例得到的梯度结果与预期一致

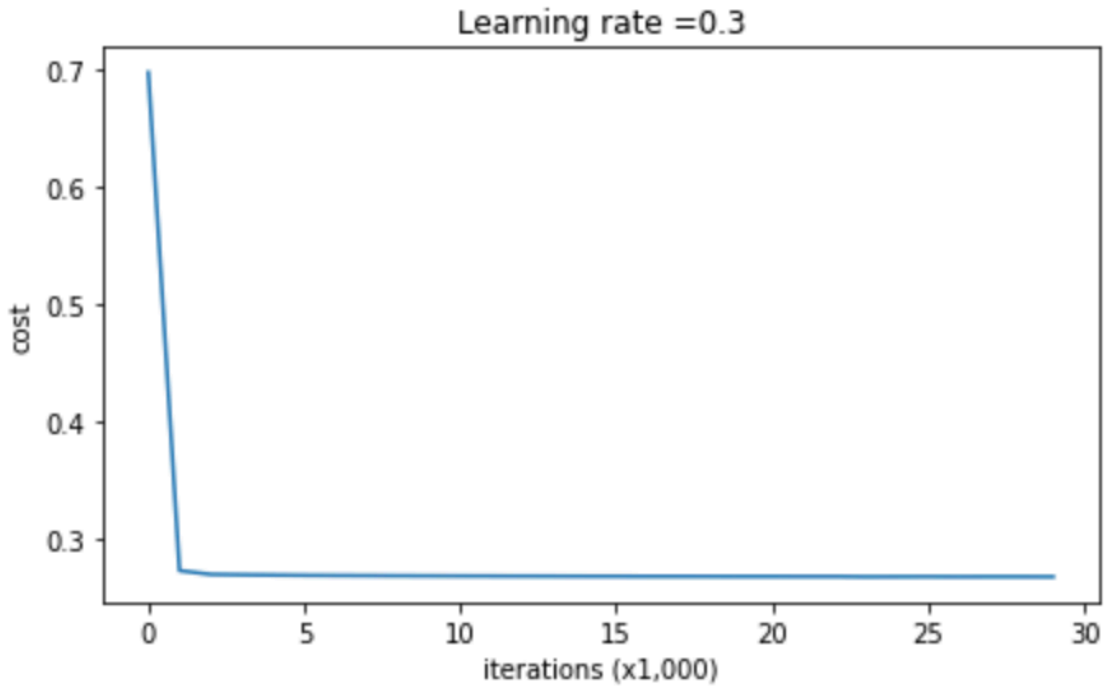
```
dW1 = [[-0.25604646  0.12298827 -0.28297129]
 [-0.17706303  0.34536094 -0.4410571 ]]
dW2 = [[ 0.79276486  0.85133918]
 [-0.0957219  -0.01720463]
 [-0.13100772 -0.03750433]]
dW3 = [[-1.77691347 -0.11832879 -0.09397446]]
```

Expected Output:

dW1	[[-0.25604646 0.12298827 -0.28297129] [-0.17706303 0.34536094 -0.4410571]]
dW2	[[0.79276486 0.85133918] [-0.0957219 -0.01720463] [-0.13100772 -0.03750433]]
dW3	[[-1.77691347 -0.11832879 -0.09397446]]

接下来使用 $(\lambda = 0.7)$的 L2正则对于数据集进行训练

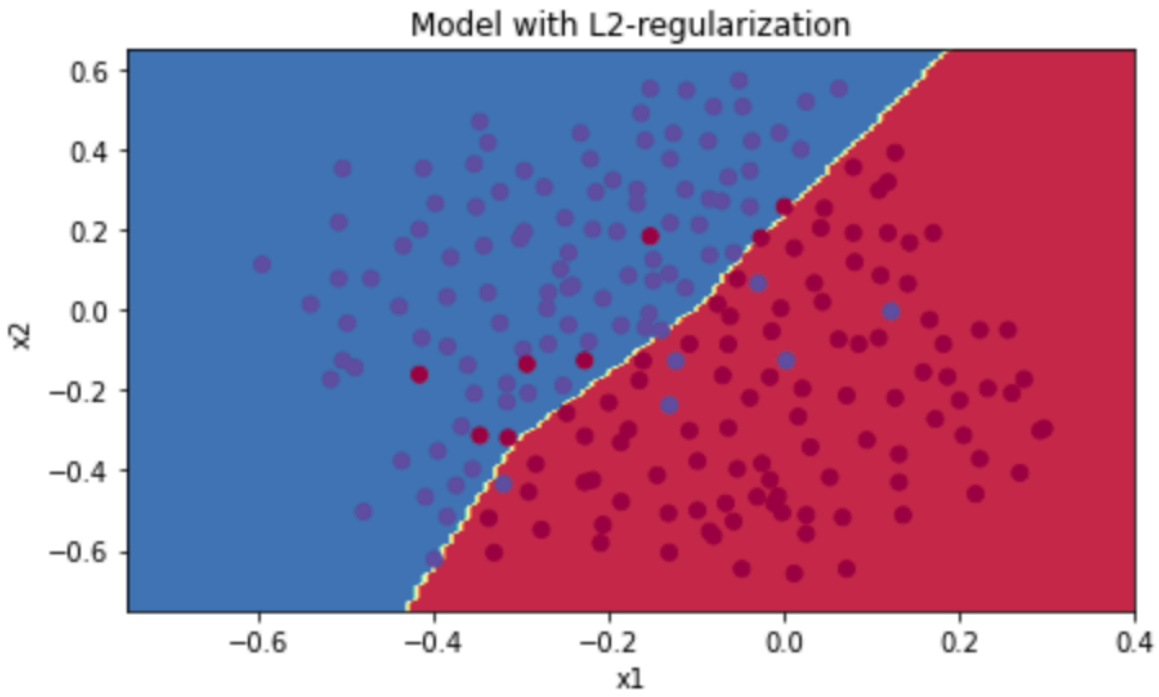
```
Cost after iteration 0: 0.6974484493131264
Cost after iteration 10000: 0.2684918873282239
Cost after iteration 20000: 0.2680916337127301
```




```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

训练集准确率:93.8%
测试集准确率:93%

通过在损失函数中加入正则项来减少噪声数据对于模型过拟合的风险，我们看到了模型在测试集上显著的性能提升
L2 正则可以使得我们得到的决策边界更加平滑



L2正则依赖于这样的假设：权重小的模型要比权重大的模型具有更大的权重。通过将成本函数中权值的平方值作为惩罚，我们可以将所有权值转换为更小的值。

3 - Dropout

3.1 - Forward propagation with dropout

`dropout` 是一种广泛使用的正则化技术
由于我们本次实验使用的是三层神经网络，所以我们在第一个和第二个隐藏层加入dropout，我们不选择在输入和输出层加入dropout。

我们每次迭代都会随机关闭第一、二个隐藏层的一些神经元

执行分为4个步骤：

- 使用 `np.random.rand()` 来随机得到一些介于0和1之间的数值来创建$a^{[1]}$同样大小的变量$d^{[1]}$，这里我们需要矩阵化实现，所以需要创建一个和$A^{[1]}$一样规格的矩阵
$D^{[1]} = [d^{[1](1)} d^{[1](2)} \dots d^{[1](m)}]$
- 通过对$D^{[1]}$中的值进行适当的阈值化，使得$D^{[1]}$中的每个条目设置为0 的概率变为(' 1–keep_prob ')或条目为1 (' keep_prob ')的概率设置为 ('keep_prob ')

- 将$A^{[1]}$设置为$A^{[1]} * D^{[1]}$。(关闭一些神经元)。可以将$D^{[1]}$看作一个掩码，这样当它与另一个矩阵相乘时，它会关闭一些值
- $A^{[1]}$除以' keep_prob '。通过这样做，您可以确保损失函数的结果仍然具有与没有dropout时有相同的期望值

step1 && step2

```
D1 = np.random.rand(A1.shape[0],(A1.shape[1])     # Step 1: initialize matrix D1 = np.random.rand(..., ...)
D1 = np.where(D1 <= keep_prob, 1, 0)              # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
```

step3 && step4

```
A1 = A1 * D1                                      # Step 3: shut down some neurons of A1
A1 = A1 / keep_prob                               # Step 4: scale the value of neurons that haven't been shut down
```

使用测试样例得到的结果：

```
A3 = [[0.36974721 0.00305176 0.04565099 0.49683389 0.36974721]]
```

Expected Output:

```
</tr>
```

A3 [[0.36974721 0.00305176 0.04565099 0.49683389 0.36974721]]

与预期保持一致

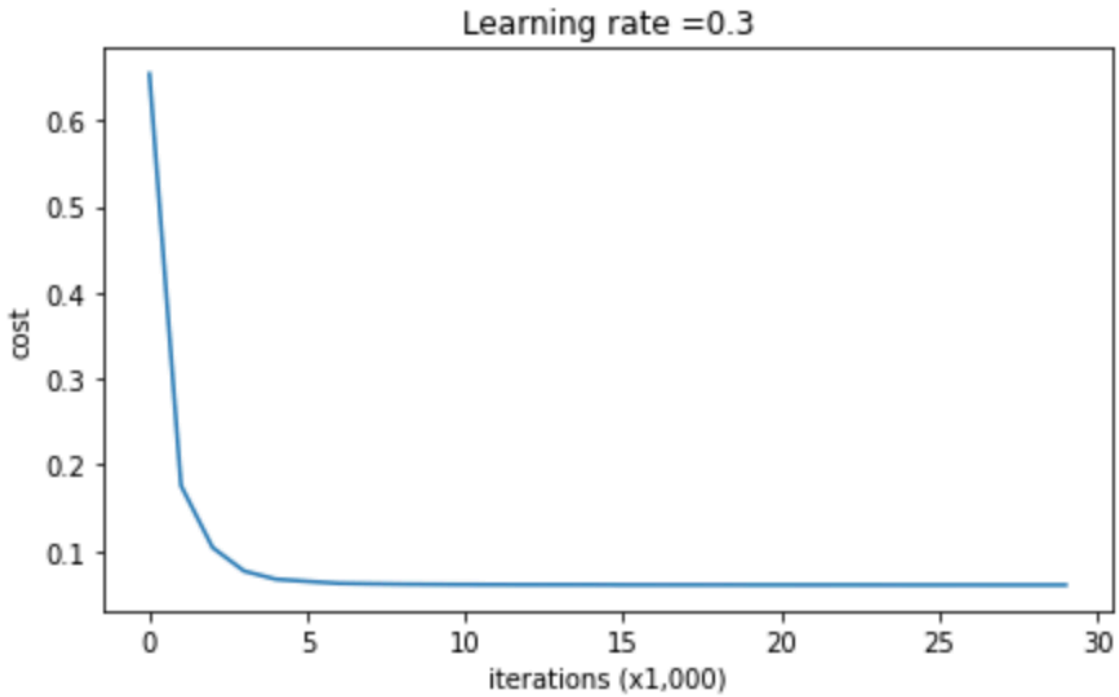
3.2 - Backward propagation with dropout

使用dropout修改后向传播，我们需要用到存储在Cache中的$D^{[1]}$ and $D^{[2]}$
我们只需要执行两个步骤：

- 1.由于在前向传播中我已经关闭了一些神经元，所以在后向传播中我需要关闭相同的神经元，将相同的mask $D^{[1]}$ 作用到 `dA1`
- 我们需要再次用 `dA1` 去除以 `keep_prob`

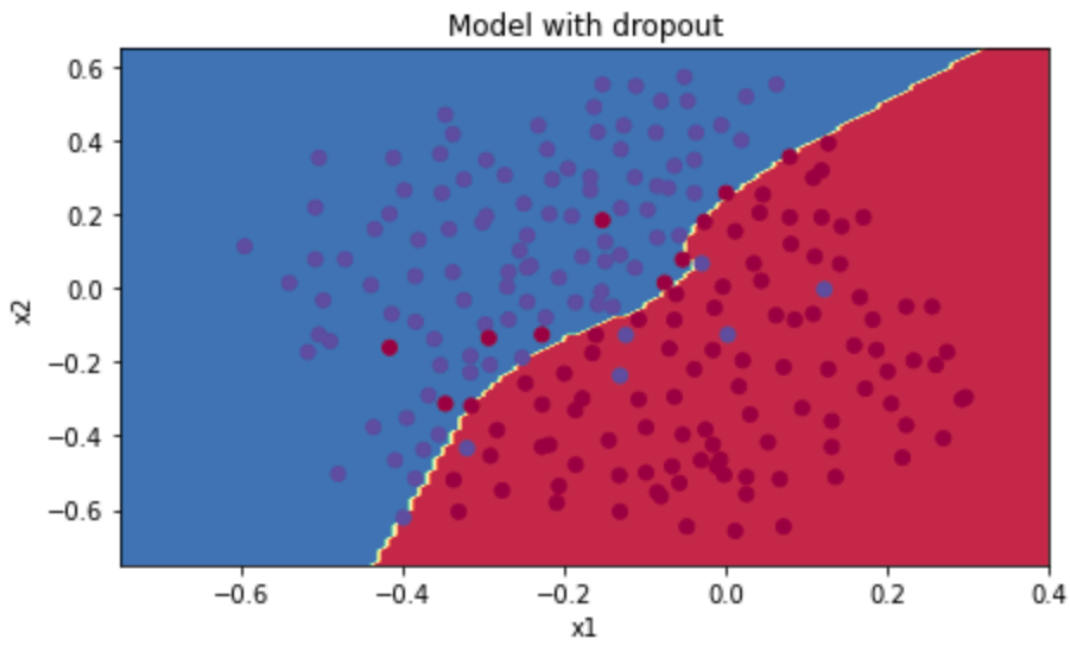
```
dA1 = [[ 0.36544439  0.         -0.00188233  0.         -0.17408748]
 [ 0.65515713  0.         -0.00337459  0.         -0.         ]]
dA2 = [[ 0.58180856  0.         -0.00299679  0.         -0.27715731]
 [ 0.          0.53159854 -0.          0.53159854 -0.34089673]
 [ 0.          0.         -0.00292733  0.         -0.         ]]
```

与预期一致




```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```

训练集准确率:92.8%
测试集准确率:95%



使用dropout时常见的错误是在训练和测试中使用它。你只能在训练中使用dropout（随机消除节点）

二.Batch Normalization

使深层网络更容易训练的一种方法是使用更复杂的优化过程，如SGD+momentum、RMSProp或Adam。另一个策略是改变网络的结构，使其更容易培训。其中一个想法是由[1]在2015年提出的批量标准化。

这个想法相对简单。当输入数据由零均值和单位方差的不相关特征组成时，机器学习方法的效果往往更好。在训练神经网络时，我们可以先对数据进行预处理，然后再将其输入网络，以显式地去关联其特征；这将确保网络的第一层能够看到遵循良好分布的数据。然而，即使我们对输入数据进行预处理，网络较深层的激活也可能不再是去相关的，也不再是零均值或单位方差，因为它们是从网络较早的层中输出的。更糟糕的是，在训练过程中，网络每一层的特征分布会随着每一层权重的更新而改变。

[1]的作者假设，深层神经网络内部特征分布的变化可能使深层神经网络的训练更加困难。为了克服这个问题，[1]建议将批处理规范化层插入到网络中。在训练时，批处理归一化层使用小批数据估计每个特征的均值和标准差。这些估计的平均值和标准偏差然后用于中心和标准化的特征的小批量。在训练期间，这些平均值和标准偏差的运行平均值被保持，在测试时，这些运行平均值被用来集中和标准化特征。

这种归一化策略可能会降低网络的表示能力，因为有时某些层的特征不是零均值或单位方差的，这可能是最优的。为此，批处理规范化层包括每个特征维的可学习的移位和缩放参数。

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

```
Batch normalization: forward
```

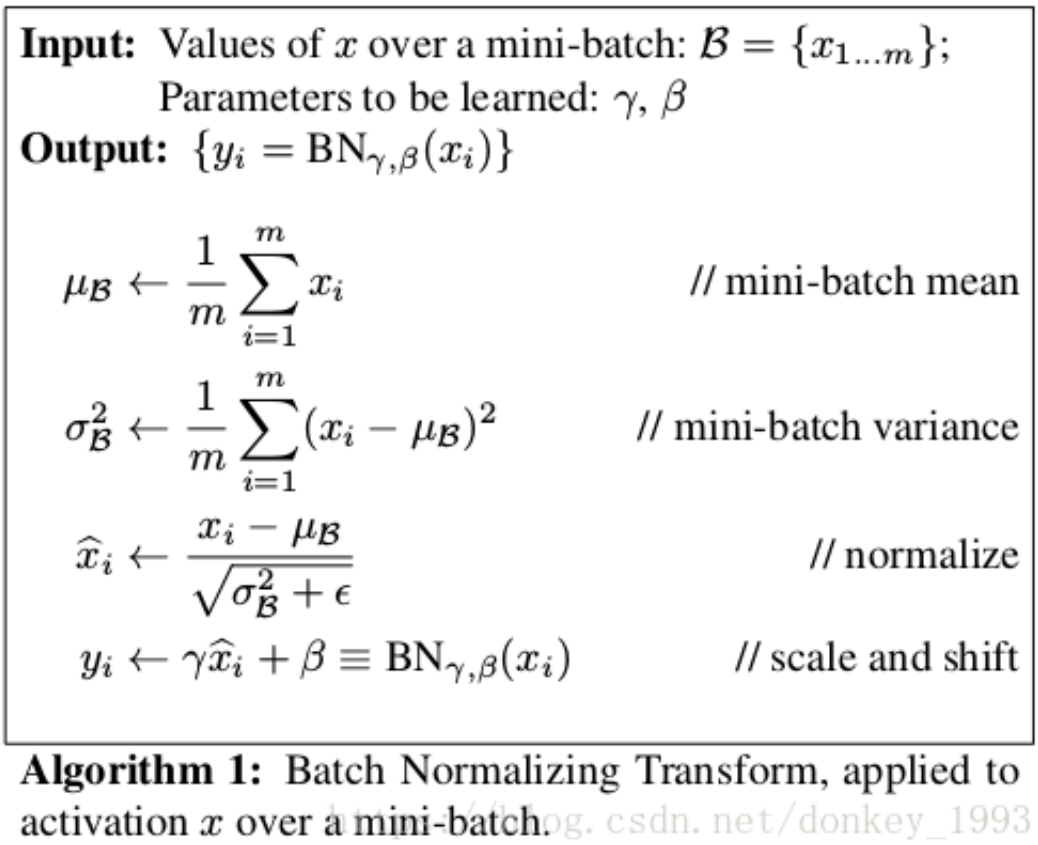
在函数 `batchnorm_forward` 中实现批处理规范化前向传递
在训练中，样本均值和(未校正的)样本方差是
从minibatch统计数据计算并用于规范化输入数据。
在训练中，我们也保持对于每个特征的均值和方差进行指数衰减的运行办法，用这些平均值进行归一化处理
测试时间的数据。

在每一步中，我们更新平均和方差的运行平均值
基于动量参数的指数衰减:

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

注意到batch normalization的论文建议了不同的测试时间行为
他们使用大量的训练图像而不是使用running average来计算每个特征的样本均值和方差。
在这次的实现中，我们选择使用running average，因为它们不需要额外的估计步骤

BN实现的伪代码：



Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

在训练的过程中我们需要

- 1.计算样本的均值和方差
- 2.然后对于样本数据进行标准化处理，这里由于我们不能让分母为0，所以在标准化的过程中引入了eps这个变量
- 3.并进行平移和缩放处理。引入了γ和β两个参数，通过训练γ和β两个参数，引入了这个可学习重构参数γ、β，让我们的网络可以学习恢复出原始网络所要学习的特征分布
- 4.根据上述公式，实现对于样本均值和样本方差的基于动量的参数衰减

```
sample_mean = np.mean(x,axis = 0) #矩阵x每一列的平均值(D,)
sample_var = np.var(x,axis = 0) #矩阵x每一列的方差(D,)
x_hat = (x - sample_mean)/(np.sqrt(sample_var + eps)) #标准化，eps:防止除数为0而增加的一个很小的正数
out = gamma * x_hat + beta #gamma放缩系数，beta偏移常量
cache = (x,sample_mean,sample_var,x_hat,eps,gamma,beta)
running_mean = momentum * running_mean + (1 - momentum) * sample_mean #基于动量的参数衰减
running_var = momentum * running_var + (1 - momentum) * sample_var
```

如果我们选择的是测试的模式
那么我们就需要使用 running 均值和方差来规范化输入数据
然后使用在训练过程中得到的γ和β的滑动均值和方差来进行数据规范化

```
elif mode == 'test':
    ##############################################################################
    # TODO: Implement the test-time forward pass for batch normalization.        #
    # Use the running mean and variance to normalize the incoming data,          #
    # then scale and shift the normalized data using gamma and beta.             #
    # Store the result in the out variable.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################
```

但是在实现 `batchnorm_forward` 之后，我进行测试的时候出现了这个错误
我查看了一下错误，感觉是我的代码文件在云平台同步的延迟导致代码不能读取
在这个点上卡顿了很久，浪费了挺多时间反复查找bug，也算是对代码有了更加深入的理解

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

After batch normalization (gamma=1, beta=0)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-c6a68b74188e> in <module>()
     14 print('After batch normalization (gamma=1, beta=0)')
     15 a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
---> 16 print_mean_std(a_norm,axis=0)
     17
     18 gamma = np.asarray([1.0, 2.0, 3.0])

<ipython-input-9-3c5d5eb57f7c> in print_mean_std(x, axis)
      9
     10 def print_mean_std(x,axis=0):
---> 11     print('  means: ', x.mean(axis=axis))
     12     print('  stds:  ', x.std(axis=axis))
     13     print()

AttributeError: 'NoneType' object has no attribute 'mean'
```

发现在训练阶段means接近于beta，stds接近于gamma
在测试阶段，仍然是means接近1，stds接近于0，相比训练阶段差别更大

Batch normalization: backward

我们需要在函数 `batchnorm_backward` 中实现对于batch normalization的后向传播
我需要出batch normalization的计算图，并通过每个中间节点进行支持
一些中间体可能有多个传出分支，我们需要确保在后向遍历中对这些分支的梯度求和

对于

```
y=γ·x̂ +β
```

其中 y 形如(N, D)，γ 和 β 形如(D,)，x̂ 形如(N, D)，所以 dβ必然为
`dbeta = np.sum(dout, axis=0)`

dy 和 dx̂ 都形如(N, D)，而 dγ 形如(D,)，显然 dγ 应为
`dgamma = np.sum(xhat * dout, axis=0)`

$$\begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1D} \\ y_{21} & y_{22} & \cdots & y_{2D} \\ & \cdots & \cdots & \\ y_{N1} & y_{N2} & \cdots & y_{ND} \end{bmatrix} = \begin{bmatrix} \gamma_1 & \gamma_2 & \cdots & \gamma_D \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ & \cdots & \cdots & \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix}$$

展开：

$$\begin{aligned} y_{11} &= \gamma_1 \cdot x_{11}, & y_{12} &= \gamma_2 \cdot x_{12}, & \cdots \\ y_{21} &= \gamma_1 \cdot x_{21}, & y_{22} &= \gamma_1 \cdot x_{22}, & \cdots \end{aligned}$$

由此我们可以得到：

$$\frac{\partial \mathrm{L}}{\partial \gamma_q} = \frac{\partial \mathrm{L}}{\partial y} \cdot \frac{\partial y}{\partial \gamma_q} = \sum_{i,j} \frac{\partial \mathrm{L}}{\partial y_{ij}} \cdot \frac{\partial y_{ij}}{\partial \gamma_q}$$

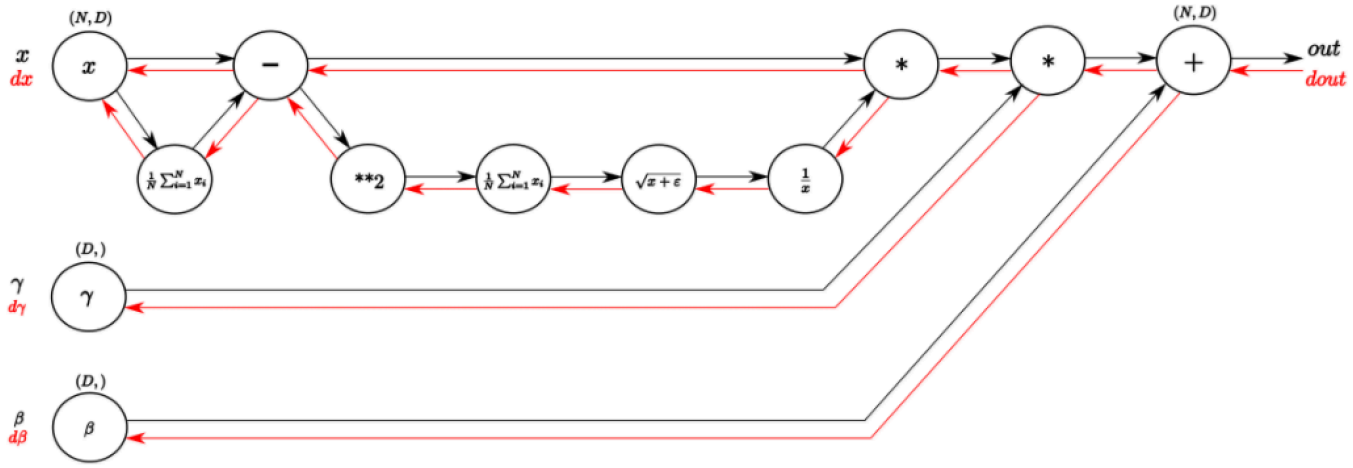
当j=q的时候，我们有

$$\frac{\partial \mathrm{L}}{\partial \gamma_q} = \sum_{i=1}^{N} \frac{\partial \mathrm{L}}{\partial y_{iq}} \cdot \frac{\partial y_{iq}}{\partial \gamma_q} = \sum_{i=1}^{N} x_{iq} \cdot \mathrm{d}y_{iq}$$

所以由公式我们可以写出如下后向的表达

```
#利用公式计算梯度
 x,mean,var,x_hat,eps,gamma,beta = cache
 N = x.shape[0]
 dgamma = np.sum(dout * x_hat,axis = 0)
 dbeta = np.sum(dout * 1.0,axis = 0)
 dx_hat = dout * gamma
 dx_hat_numerator = dx_hat / np.sqrt(var + eps)
 dx_hat_denominator = np.sum(dx_hat * (x - mean),axis = 0)
 dx_1 = dx_hat_numerator
 dvar = -0.5 * ((var + eps) ** (-1.5)) * dx_hat_denominator
 dmean = -1.0 * np.sum(dx_hat_numerator,axis = 0) + dvar * np.mean(-2.0 * (x - mean),axis = 0)
 dx_var = dvar * 2.0 / N * (x - mean)
 dx_mean = dmean * 1.0 / N
 dx = dx_1 + dx_var + dx_mean
```

计算图：



Batch normalization: alternative backward

首先，我们计算过程中的中间变量：

$$
\begin{aligned}
\mathrm{d}out &= \frac{\partial L}{\partial y} \\
y &= \gamma \cdot \hat{x} + \beta \\
\hat{x} &= \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \\
\mu &= \frac{1}{N} \sum_{n=1}^{N} x_n \\
\sigma^2 &= \frac{1}{N} \sum_{n=1}^{N} (x_n - \mu)^2
\end{aligned}
$$

计算对 x_{ij} 的导数：

$$
\begin{aligned}
\frac{\partial L}{\partial x_{ij}} &= \sum_{n,d} \frac{\partial L}{\partial y_{nd}} \cdot \frac{\partial y_{nd}}{\partial x_{ij}} \\
&= \sum_{n,d} \frac{\partial L}{\partial y_{nd}} \cdot \frac{\partial y_{nd}}{\partial \hat{x}_{nd}} \cdot \frac{\partial \hat{x}_{nd}}{\partial x_{ij}}
\end{aligned}
$$

其中:

$$
\begin{aligned}
y_{nd} &= \gamma_d \cdot \hat{x_{nd}} + \beta_d \\
\hat{x_{nd}} &= \frac{x_{nd} - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}} \\
\mu_d &= \frac{1}{N} \sum_{n=1}^{N} x_{nd} \\
\sigma_d^2 &= \frac{1}{N} \sum_{n=1}^{N} (x_{nd} - \mu_d)^2 \\
\frac{\partial y_{nd}}{\partial \hat{x_{nd}}} &= \gamma_d
\end{aligned}
$$

下面的工作就是要计算 $\frac{\partial \hat{x_{nd}}}{\partial x_{ij}}$：

$$
\begin{aligned}
\frac{\partial \hat{x_{nd}}}{\partial x_{ij}} &= \frac{\partial}{\partial x_{ij}} \left(\frac{x_{nd} - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}} \right) \\
&= \left(\sigma_d^2 + \epsilon\right)^{-\frac{1}{2}} \cdot \frac{\partial}{\partial x_{ij}} (x_{nd} - \mu_d) + (x_{nd} - \mu_d) \cdot \frac{\partial}{\partial x_{ij}} \left(\sigma_d^2 + \epsilon\right)^{-\frac{1}{2}} \\
&= \left(\sigma_d^2 + \epsilon\right)^{-\frac{1}{2}} \cdot \frac{\partial}{\partial x_{ij}} (x_{nd} - \mu_d) - \frac{1}{2} \left(\sigma_d^2 + \epsilon\right)^{-\frac{3}{2}} (x_{nd} - \mu_d) \cdot \frac{\partial \sigma_d^2}{\partial x_{ij}}
\end{aligned}
$$

接下来，我们计算：

$$
\begin{aligned}
\frac{\partial}{\partial x_{ij}} (x_{nd} - \mu_d) &= \frac{\partial}{\partial x_{ij}} \left(x_{nd} - \frac{1}{N} \sum_{t=1}^{N} x_{td} \right) \\
&= \frac{\partial x_{nd}}{\partial x_{ij}} - \frac{1}{N} \frac{\partial}{\partial x_{ij}} \left(\sum_{t=1}^{N} x_{td} \right)
\end{aligned}
$$

第一项，当且仅当 n=i, d=j 时不为0，第二项中仅有 d=j 项不为0，故：

$$
\frac{\partial}{\partial x_{ij}} (x_{nd} - \mu_d) = \delta_{n,i} \cdot \delta_{d,j} - \frac{1}{N} \delta_{d,j}
$$

接着计算：

$$
\begin{aligned}
\frac{\partial \sigma_d^2}{\partial x_{ij}} &= \frac{\partial}{\partial x_{ij}} \left(\frac{1}{N} \sum_{n=1}^{N} (x_{nd} - \mu_d)^2 \right) \\
&= \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial x_{ij}} \left((x_{nd} - \mu_d)^2 \right) \\
&= \frac{2}{N} \sum_{n=1}^{N} (x_{nd} - \mu_d) \frac{\partial}{\partial x_{ij}} (x_{nd} - \mu_d) \\
&= \frac{2}{N} \sum_{n=1}^{N} (x_{nd} - \mu_d) \cdot \left(\delta_{n,i} \cdot \delta_{d,j} - \frac{1}{N} \delta_{d,j} \right) \\
&= \frac{2}{N} \sum_{n=1}^{N} (x_{nd} - \mu_d) \cdot \delta_{n,i} \cdot \delta_{d,j} - \frac{2}{N^2} \sum_{n=1}^{N} (x_{nd} - \mu_d) \cdot \delta_{d,j}
\end{aligned}
$$

第一项中，仅有 n=i 一项不为0：

$$\sum_{n=1}^{N}(x_{nd}-\mu_d)\cdot\delta_{n,i}\cdot\delta_{d,j}=(x_{id}-\mu_d)\cdot\delta_{d,j}$$

第二项：

$$\sum_{n=1}^{N}(x_{nd}-\mu_d)=\sum_{n=1}^{N}x_{nd}-N\cdot\mu_d$$

但因为$\mu_d=\frac{1}{N}\sum_{n=1}^{N}X_{nd}$,因此上式为0

$$\frac{\partial\sigma_d^2}{\partial x_{ij}}=\frac{2}{N}(x_{id}-\mu_d)\cdot\delta_{d,j}$$

综上：

$$\begin{aligned}\frac{\partial\hat{x_{nd}}}{\partial x_{ij}}&=\left(\sigma_d^2+\epsilon\right)^{-\frac{1}{2}}\cdot\frac{\partial}{\partial x_{ij}}(x_{nd}-\mu_d)-\frac{1}{2}\left(\sigma_d^2+\epsilon\right)^{-\frac{3}{2}}(x_{nd}-\mu_d)\cdot\frac{\partial\sigma_d^2}{\partial x_{ij}}\\&=\left(\sigma_d^2+\epsilon\right)^{-\frac{1}{2}}\cdot\left(\delta_{n,i}\cdot\delta_{d,j}-\frac{1}{N}\delta_{d,j}\right)-\frac{1}{N}\left(\sigma_d^2+\epsilon\right)^{-\frac{3}{2}}(x_{nd}-\mu_d)(x_{id}-\mu_d)\cdot\delta_{d,j}\end{aligned}$$

最后，计算对 xij 的导数：

$$\begin{aligned}\frac{\partial L}{\partial x_{ij}}&=\sum_{n,d}\frac{\partial L}{\partial y_{nd}}\cdot\gamma_d\cdot\frac{\partial\hat{x_{nd}}}{\partial x_{ij}}\\&=\sum_{n,d}\gamma_d\cdot\frac{\partial L}{\partial y_{nd}}\cdot\left(\sigma_d^2+\epsilon\right)^{-\frac{1}{2}}\cdot\left(\delta_{n,i}\cdot\delta_{d,j}-\frac{1}{N}\delta_{d,j}\right)-\frac{1}{N}\sum_{n,d}\gamma_d\cdot\frac{\partial L}{\partial y_{nd}}\cdot\left(\sigma_d^2+\epsilon\right)^{-\frac{3}{2}}(x_{nd}-\mu_d)(x_{id}-\mu_d)\cdot\delta_{d,j}\end{aligned}$$

第一项：

$$\begin{aligned}&\sum_{n,d}\gamma_d\cdot\frac{\partial L}{\partial y_{nd}}\cdot\left(\sigma_d^2+\epsilon\right)^{-\frac{1}{2}}\cdot\left(\delta_{n,i}\cdot\delta_{d,j}-\frac{1}{N}\delta_{d,j}\right)\\&=\sum_{n}\gamma_j\cdot\frac{\partial L}{\partial y_{nj}}\cdot\left(\sigma_j^2+\epsilon\right)^{-\frac{1}{2}}\cdot\left(\delta_{n,i}-\frac{1}{N}\right)\\&=\gamma_j\cdot\frac{\partial L}{\partial y_{ij}}\cdot\left(\sigma_j^2+\epsilon\right)^{-\frac{1}{2}}-\frac{1}{N}\sum_{n}\gamma_j\cdot\frac{\partial L}{\partial y_{nj}}\cdot\left(\sigma_j^2+\epsilon\right)^{-\frac{1}{2}}\\&=\gamma_j\cdot\left(\sigma_j^2+\epsilon\right)^{-\frac{1}{2}}\left(\frac{\partial L}{\partial y_{ij}}-\frac{1}{N}\sum_{n}\frac{\partial L}{\partial y_{nj}}\right)\end{aligned}$$

第二项：

$$
\begin{aligned}
&-\frac{1}{N}\sum_{n,d}\gamma_d\cdot\frac{\partial L}{\partial y_{nd}}\cdot\left(\sigma_d^2+\epsilon\right)^{-\frac{3}{2}}\left(x_{nd}-\mu_d\right)\left(x_{id}-\mu_d\right)\cdot\delta_{d,j}\\
=&-\frac{1}{N}\sum_{n}\gamma_j\cdot\frac{\partial L}{\partial y_{nj}}\cdot\left(\sigma_j^2+\epsilon\right)^{-\frac{3}{2}}\left(x_{nj}-\mu_j\right)\left(x_{ij}-\mu_j\right)\\
=&-\frac{1}{N}\gamma_j\left(\sigma_j^2+\epsilon\right)^{-\frac{3}{2}}\left(x_{ij}-\mu_j\right)\sum_{n}\frac{\partial L}{\partial y_{nj}}\left(x_{nj}-\mu_j\right)
\end{aligned}
$$

最后的结果为：

$$
\frac{\partial L}{\partial x}=\frac{\gamma}{N}\left(\sigma^2+\epsilon\right)^{-\frac{1}{2}}\left(N\frac{\partial L}{\partial y}-\sum_n\frac{\partial L}{\partial y_n}-\left(\sigma^2+\epsilon\right)^{-1}(x-\mu)\sum_n\frac{\partial L}{\partial y_n}(x_n-\mu)\right)
$$

所以我们的代码实现：

```
x,mean,var,x_hat,eps,gamma,beta = cache
N = x.shape[0]
dbeta = np.sum(dout,axis = 0)
dgamma = np.sum(x_hat * dout,axis = 0)
dx_norm = dout * gamma
dv = ((x - mean) * -0.5 * (var + eps)**-1.5 * dx_norm).sum(axis = 0)
dm = (dx_norm * -1 * (var + eps)** -0.5).sum(axis = 0) + (dv * (x - mean) * -2 / N).sum(axis = 0)
dx = dx_norm / (var + eps) ** 0.5 + dv * 2 * (x - mean) / N + dm / N
```

结果很尴尬，还不如不加速。。

结论和体会：
这次实验实现了正则化和BN的上手体会，相比正则化，可以说BN的推导过程要复杂不少，但是还是看到了这两个优化的办法在提高神经网络性能的优势和普适性。