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：Regularization and Batch Normalization		学号：202000130047
日期：2022-10-15	班级： 智能 20	姓名： 夏再禹
Email：842649082@qq.com		

实验目的：
对优化神经网络的两种方法进行实验：Regularization and Batch Normalization。补充编写完实验所需代码，通过结果体会这两种方法的效果。

实验软件和硬件环境：
软件：jupyter notebook
硬件：cpu: Intel i5

实验原理和方法：
1. Batch normalization

一般来说，当输入数据的均值和方差为零，且各维度的特征不相关时，机器学习方法会工作得更好。

对数据集的预处理只能确保网络的第一层可以有良好分布的数据，但深层网络的数据的分布就无法保证了。而且在训练过程中，网络每一层的特征分布都会随着每一层权重的更新而改变。

因此如果想要在深层网络也保证良好的数据分布，就需要特殊的结构，于是就有了Batch normalization。

Batch normalization layer 使用 minibatch 的数据来估计每个特征的平均值和标准差，然后使用这些估计值来 normalize 这个 minibatch 的数据。

此外，考虑到 test 过程的处理，需要在 train 过程中计算平均值与标准差的 running average，在 test 过程中，使用这些 running average 来 normalize 特征。

训练过程中的算法如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

2. Regularization

正则化是用来防止过拟合的手段，我们这次实验 L2- Regularization 与 dropout 两种正则化方法。

L2- Regularization 通过在损失函数中加上对参数的平方和的惩罚，来使不重要的参数值的绝对值尽量小，从而减小过拟合。

Dropout 即在每次迭代训练时，随机地使一些神经元失活。其原理为，这样可以降低神经元对其它特定神经元被激活的敏感性，因为其它的神经元可能随时被失活，从而使得神经元有更强的独立性。

实验步骤：（不要求罗列完整源代码）

一、Batch normalization

根据论文中的训练算法，补充前向传播的代码如下：

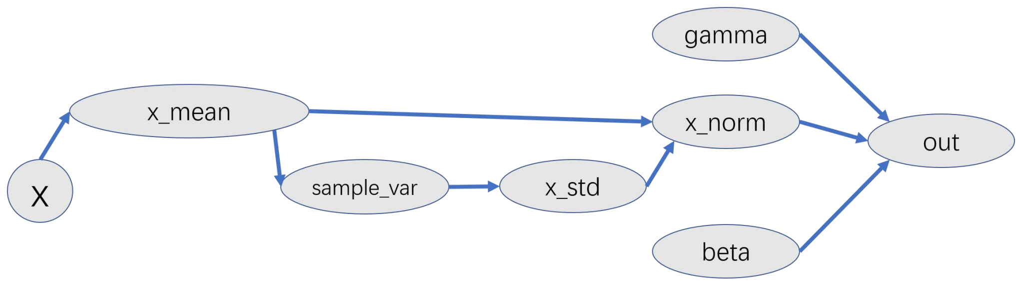
Train:

```
# compute the mean and variance of x
sample_mean = np.mean(x,axis=0)
x_mean = x-sample_mean
sample_var = np.average(x_mean**2,axis=0)
x_std = np.sqrt(sample_var+eps)
# normalize
x_norm = x_mean/x_std
# scale and shift
out = gamma*x_norm + beta
# store for the cache
cache = (x,x_mean,sample_var,eps,x_std,x_norm,gamma)
# update the running mean and running variable
running_mean = momentum * running_mean + (1 - momentum) * sample_mean
running_var = momentum * running_var + (1 - momentum) * sample_var
```

Test:

```
running_std = np.sqrt(running_var+eps)
x = (x-running_mean)/running_std
# scale and shift
out = gamma*x + beta
```

对于反向传播，考虑我们在上述过程中的计算图：



故对于 x 的梯度 dx,我们可以从 x_norm 的梯度 dx_norm 依次求到 dx_std、dsample_var、dx_mean，最后即可算出 dx。计算代码如下：

```
N,_ = dout.shape
x,x_mean,sample_var,eps,x_std,x_norm,gamma = cache
dx_norm = gamma*dout
dx_std = np.sum(-x_mean/(x_std**2)*dx_norm,axis=0)
dsample_var = 0.5/np.sqrt(sample_var+eps)*dx_std
dx_mean = (1./x_std) * dx_norm + 2./N*x_mean*dsample_var
dx = dx_mean - 1./N*np.sum(dx_mean,axis=0)
dgamma = np.sum(x_norm*dout,axis=0)
dbeta = np.sum(dout,axis=0)
```

梯度检验的输出如下，说明反向传播代码正确。

```
dx error:  1.7029235612572515e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

对于反向传播代码的优化，我们考虑舍弃中间过程 dx_std、dsample_var，直接从 dout 推导出 dx_norm，进而推导出 dmean 再推导出 dx，通过手算进行化简。
最后推导出了如下代码：

```
N,_ = dout.shape
x,x_mean,sample_var,eps,x_std,x_norm,gamma = cache
dx_norm = gamma*dout
temps =  np.sum(x_mean*dx_norm,axis=0)/(sample_var+eps)/N
dx_mean = dx_norm/x_std - x_norm*temps
dx = dx_mean - np.sum(dx_mean,axis=0)/N
dgamma = np.sum(x_norm*dout,axis=0)
dbeta = np.sum(dout,axis=0)
```

验证比较两种算法的计算速度：

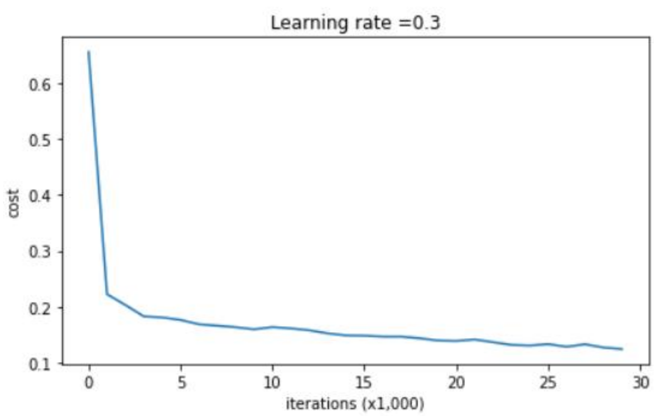
```
dx difference:  3.0666812374059947e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.84x
```

由于运行时缓存等影响，时间会变化很大，有的时候上述代码会发生除零异常，因为其记录 batchnorm_backward_alt()函数的运行时间为 0，这可能是缓存导致的。所以我运行了很多次，取了一个比较有代表性的结果如上图，加速了 1.84x。实际上加速比例很难用单次函数运行衡量，加速的值有的时候很大，有的时候小于 1，即有“减速“。

二.　　Regularization

1.　无 Regularization

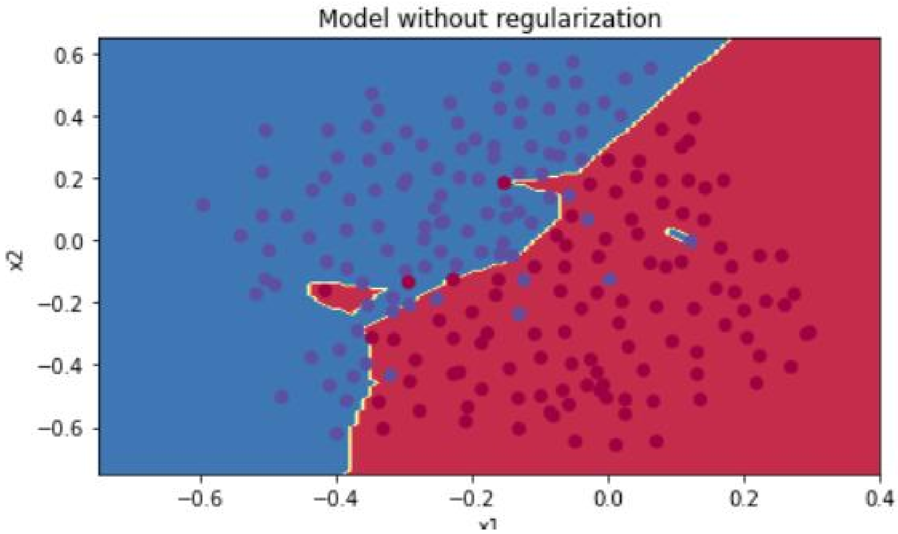
loss 的变化情况为：



准确度为：

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

分类边界为：



2. L2- regularization

L2- regularization 就是在损失函数中加上一个参数的平方和的倍数，超参λ用来控制正则项加入的比例。

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

To:

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

补全代码：

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

测试一个损失函数的值如下，与预期相同，证明代码正确。

```
cost = 1.7864859451590758
```

对 L2 正则项的反向传播，我们需要对其求导：

$$\frac{d}{dW}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right) = \frac{\lambda}{m}W$$

由此，我们只需要在反向传播计算梯度时，对于 dW 加上 $\frac{\lambda}{m}W$ 即可。

补全代码：

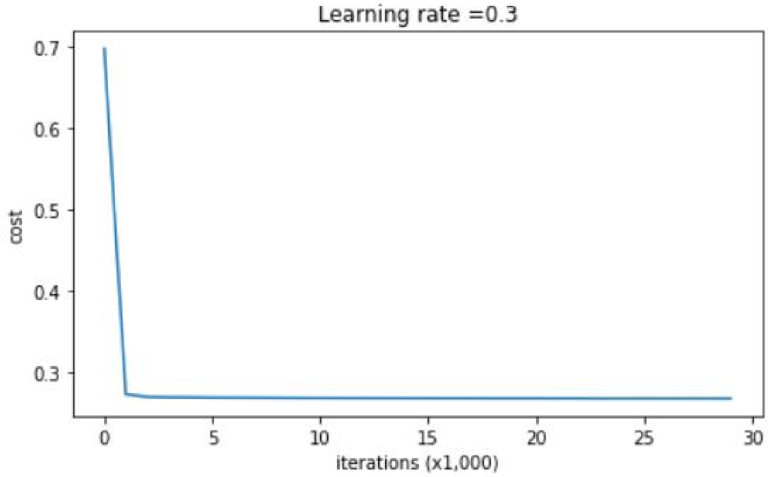
```
### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd/m)*W3
### END CODE HERE ###
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

dA2 = np.dot(W3.T, dZ3)
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
### START CODE HERE ### (approx. 1 line)
dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd/m)*W2
### END CODE HERE ###
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
### START CODE HERE ### (approx. 1 line)
dW1 = 1./m * np.dot(dZ1, X.T) + (lambd/m)*W1
### END CODE HERE ###
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
```

测试一个样例，计算梯度如下图，与预期的输出相同，证明程序正确。

```
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]]
```

于是我们用带 L2 正则项的损失来训练模型。

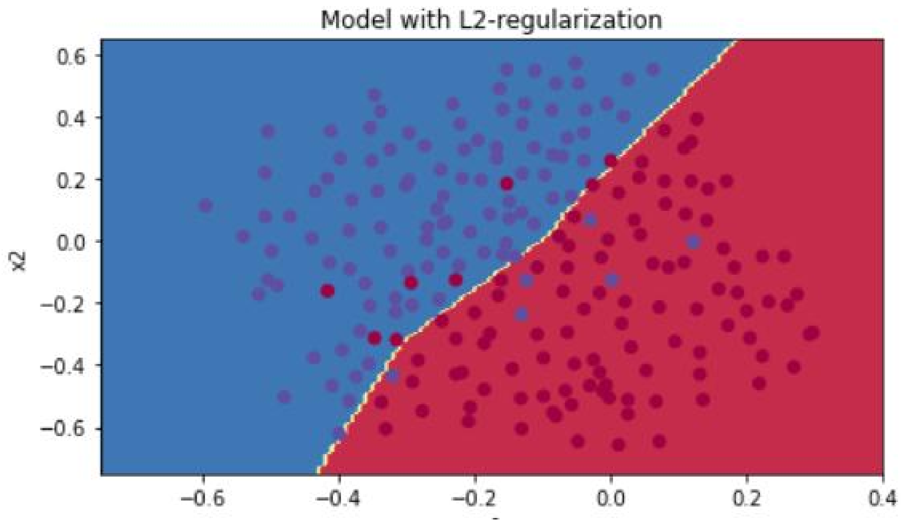
Loss 变化情况为：



准确度为：

```
On the train set:
Accuracy: 0.938388625592417l
On the test set:
Accuracy: 0.93
```

分类边界为：



可见，带 L2 正则项的模型虽然在训练集上的 accuracy 比不带的低，但其在测试集上的 accuracy 比无正则化的模型的大，所以 L2 正则项减少了模型的过拟合。

从分类边界上看，带 L2 正则项的模型的分类边界更光滑，且受异常数据的影响较小。

3. dropout

每次以一个 1-keep_prob 的概率使神经元失活，补全代码如下：

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

输出结果如下图，与预期输出相同。

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

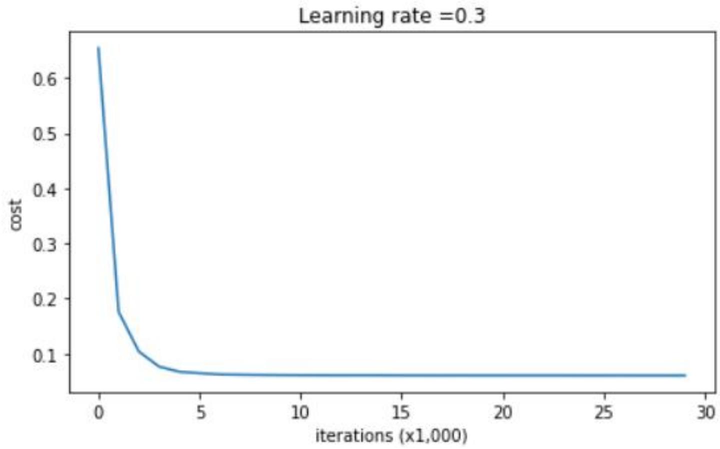
Dropout 的反向传播：

对梯度也乘以 D2,使失活的神经元梯度为 0 即可

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

训练结果：

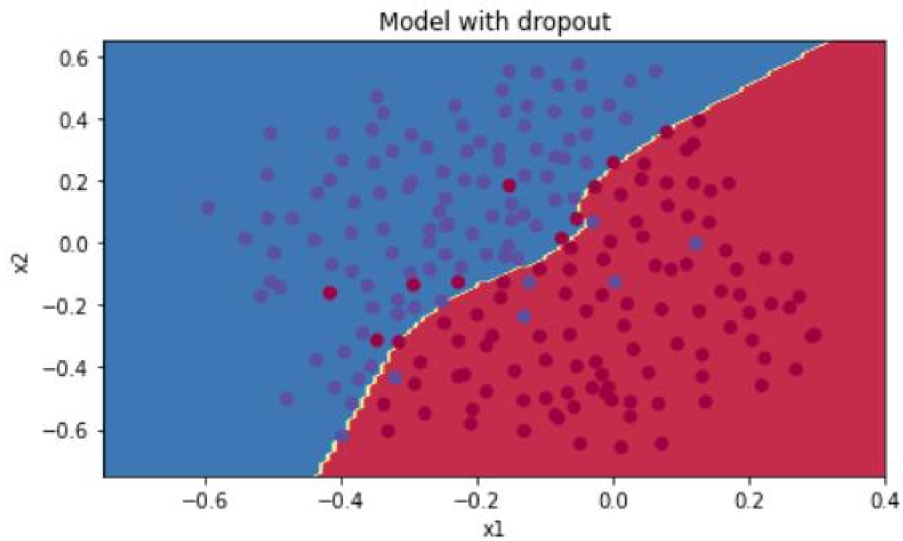
Loss 变化情况为：



准确度为：

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

分类边界为：



可见在测试集上的 accuracy 甚至比在训练集上的还高，减少过拟合的程度更高。
观察其分类边界，可见使用 dropout 的模型比未正则化的模型的分类边界更光滑，且受异常数据的影响较小。

三种模型的 accuracy 汇总如下：

model	**train accuracy**	**test accuracy**
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

可见 dropout 在测试集上的 accuracy 最好，这三种模型 train accuracy 降低时，test accuracy 升高，说明过拟合的现象普遍存在。

结论分析与体会：

通过本次实验，体会了 Batch normalization 与正则化中的 L2 正则化及 dropout 如何实现，不仅完成了其前向传播过程，也完成了反向传播过程。对他们如何实现，以及其中的细节有了更深的体会。

此外还对正则化加入与否的模型的效果进行了测试，实验证明加入正则化的效果更好，可减少过拟合，在本次实验中 dropout 加入后取得了最好的效果。

本次实验的 Batch normalization 的求导部分较有难度，特别是其涉及到“矩阵求导“（也可把矩阵拆开来看），很容易出错，这里我 debug 了很久才解决，才发现自己对于“矩阵求导“的一个很大的误区。故在本次实验中，对反向传播过程、求导过程有了更正确的认识。

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. 矩阵求导部分，一直有 bug

事实上对于输入 x，要注意其中的元素在计算图中有多个路径。可以把矩阵 x 拆成一个个向量来看，更清晰。拆开后，可画出如下的计算图，然后再用链式求导法则来计算，就不容易出错了