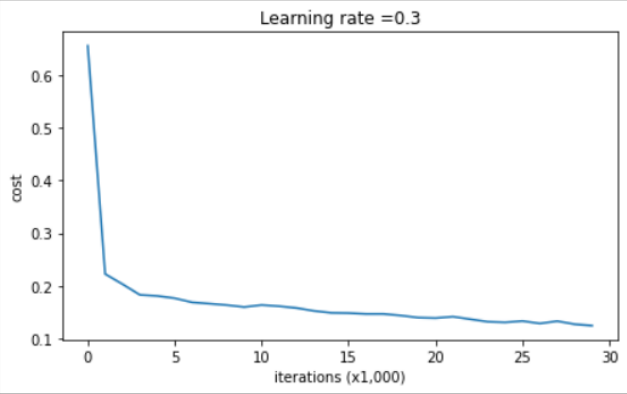
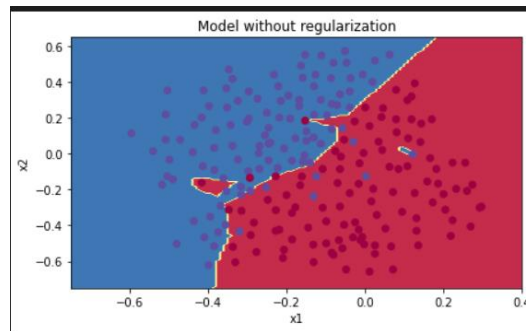


计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：神经网络的基本训练		学号：201900150221
日期：10.14	班级：19 智能	姓名：张进华
Email: zjh15117117428@163.com		
实验目的： 掌握基本的神经网络调整技巧并尝试，改进深度神经网络：超参数调整、正则化和优化，批量标准化，完成两个子任务：正则化、批量标准化。		
实验软件和硬件环境： Visual studio Code python 3.9.7 Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz		
实验原理和方法 1. 正则化是为了减少过拟合的问题，通过改变损失函数，改变学习的梯度，从而使得学习得到的网络能够适用于除了训练集以外的其他数据。 2. Batch Normalization (批归一化)，是网络中的一个层，常用在非线性层之前，先进行归一化，再对这个数据进行放缩和平移，可以提高学习率并且加快收敛速度		
实验步骤 <h2>步骤一 Regularization</h2> <h3>1 - Non-regularized model</h3> <p>首先观察一下不使用正则化下模型的效果，实现效果如下：</p> <div data-bbox="459 1352 1222 1935"><p>On the training set: Accuracy: 0.9478672985781991 On the test set: Accuracy: 0.915</p></div>		
<p>由图可以观察到在训练集和测试集上准确率都非常高，训练集精确度为 94.8%；而对于测试集，精确度为 91.5%，同时随着迭代次数的增加，cost 不断减小，最后趋于一</p>		

个很小值。绘制决策边界曲线实现效果如下：



从图中可以看出，在无正则化时，决策边界曲线有了明显的过拟合特性。

2 - L2 Regularization

L2 正则化通过适当修改成本函数来避免过度拟合，新旧成本函数的计算公式如下：

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

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

由公式编写使用正则化计算损失函数 compute_cost_with_regularization, 实现代码如下：

```
m = Y.shape[1]
W1 = parameters["W1"]
W2 = parameters["W2"]
W3 = parameters["W3"]

cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

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

cost = cross_entropy_cost + L2_regularization_cost

return cost
```

实现向后传播的函数，更新公式如下：

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

其中所有的梯度都必须根据新的成本值来计算，实现代码如下：

```

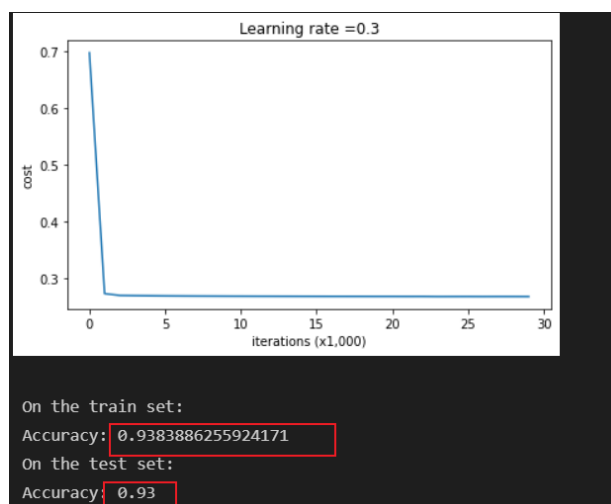
### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + ((lambda * W3) / m)
### END CODE HERE ###
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

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) + ((lambda * W2) / m)
### END CODE HERE ###
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

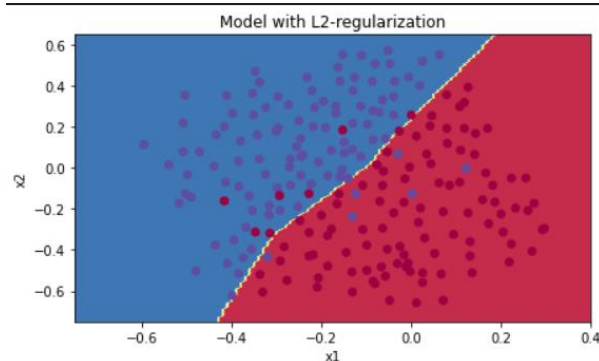
dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
### START CODE HERE ### (approx. 1 line)
dW1 = 1./m * np.dot(dZ1, X.T) + ((lambda * W1) / m)
### END CODE HERE ###
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

```

设定正则化参数 $\lambda = 0.7$, 实现预测模型效果如下:



由图可以发现使用 L2 正则化不再过度拟合训练数据, 绘制决策边界曲线如下:



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较小的值。权重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，但是需要花费更多的时间。L2 正则化对以下内容有影响：

成本计算：正则化的计算需要添加到成本函数中

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
重量变小（“重量衰减”）：权重被逐渐改变到较小的值。

3 - Dropout

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，实际上修改了模型。背后的想法是，在每次迭代时，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，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

3.1 - Forward propagation with dropout

由公式实现代码如下：

```
# LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
Z1 = np.dot(W1, X) + b1
A1 = relu(Z1)
### 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 ###
Z2 = np.dot(W2, A1) + b2
A2 = relu(Z2)
### START CODE HERE ### (approx. 4 lines)
D2 = np.random.rand(A2.shape[0],A2.shape[1])
D2 = D2 < keep_prob
A2 = A2 * D2
A2 = A2 / keep_prob
### END CODE HERE ###
Z3 = np.dot(W3, A2) + b3
A3 = sigmoid(Z3)
```

3.2 - Backward propagation with dropout

改变了前向传播的算法，也需要改变后向传播的算法，使用存储在缓存中的掩码，将舍弃的节点位置信息添加到第一个和第二个隐藏层，由公式实现代码如下：

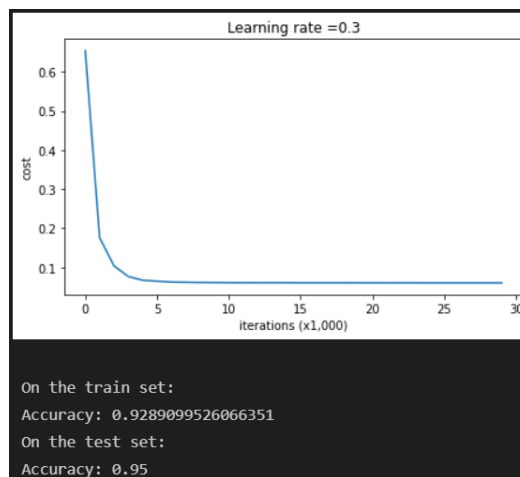
```

dZ3 = A3 - Y
dW3 = 1./m * np.dot(dZ3, A2.T)
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
dA2 = np.dot(W3.T, dZ3)
### START CODE HERE ### (≈ 2 lines of code)
dA2 = dA2 * D2 # Step 1: Apply mask D2 to shut down the same neurons
dA2 = dA2 / keep_prob # Step 2: Scale the value of neurons that haven't been
### END CODE HERE ###
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
dW2 = 1./m * np.dot(dZ2, A1.T)
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

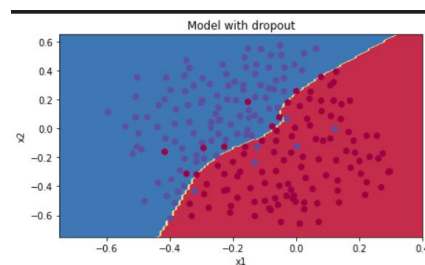
dA1 = np.dot(W2.T, dZ2)
### START CODE HERE ### (≈ 2 lines of code)
dA1 = dA1 * D1 # Step 1: Apply mask D1 to shut down the same neurons as during
dA1 = dA1 / keep_prob # Step 2: Scale the value of neurons that haven't been
### END CODE HERE ###
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
dW1 = 1./m * np.dot(dZ1, X.T)
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

```

运行带有 dropout 的模型，设置 keep_prob = 0.86，这意味着在每次迭代中，以 24% 的概率关闭第 1 层和第 2 层的每个神经元



由图可知 dropout 效果测试准确率再次提高（达到 95%）！模型没有过度拟合训练集，并且在测试集上表现出色，同时运行代码来绘制决策边界如下：



使用 dropout 时的常见错误是在训练和测试中都使用它，应该只在训练中使用 dropout（随机消除节点）

步骤二 Batch Normalization

深度神经网络内部特征的变化分布可能会使训练深度网络更加困难。为了克服这个问题，在网络中插入批量归一化层。在训练时，批量归一化层使用小批量数据来估计每个特征的均值和标准差。然后使用这些估计的均值和标准差来对小批量的特征进行中心化和归一化。在训练期间保留这些均值和标准差的运行平均值，并在测试时使用这些运行平均值对特征进行中心化和归一化

1-Batch normalization: forward

在文件 layers.py 中，在函数 batchnorm_forward 中实现批量归一化前向传递，前向传播时，每个 mini-batch 的数据都用自己的均值方差来标准化，然后在训练的过程中，会计算一个 running_mean 和 running_var 更新公式如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

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

前向过程的实现代码如下：

如果 mode = “train”，训练阶段计算的是每一个 batch 的均值和方差

```
sample_mean = np.mean(x,axis = 0) #矩阵x每一列的平均值(0,)
sample_var = np.var(x,axis = 0) #矩阵x每一列的方差(0,)
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,ga (variable) momentum: Any
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 mode = “test”，测试时用的是训练后的滑动平均（我理解也就是一种加权平均）的均值和方差

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

检查前向传递批量归一化前后的特征数量通过均值和方差来，模拟两层网络的前向传播，实现效果如下：

```
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:  [ 3.99680289e-17 -3.88578059e-17  5.84948756e-17]
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]
```

可以看到在 gamma = 1, beta = 0 的情况下，mean 接近 0, 而 stds 接近 1。然后在第二次时 means 接近 beta, 而 stds 接近 gamma。

在 test 情况下效果如下：

```
3] ✓ 0.9s
c/> After batch normalization (test-time):
      means:  [-0.03927354 -0.04349152 -0.10452688]
      stds:   [1.01531428  1.01238373  0.97819988]
```

Means 接近 0, 而 stds 接近 1

2- Batch normalization: backward

在函数 “batchnorm_backward” 中实现批量归一化的反向传递, 由论文中的公式可得：

$$\begin{aligned}\frac{\partial \ell}{\partial \hat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\ \frac{\partial \ell}{\partial \sigma_B^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2} \\ \frac{\partial \ell}{\partial \mu_B} &= \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_B)}{m} \\ \frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\partial \ell}{\partial \mu_B} \cdot \frac{1}{m} \\ \frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}\end{aligned}$$

实现代码如下：

```
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
```

梯度检查 batchnorm 向后传递

```
23 print('dbeta error: ', rel_error(db_num, dbet
[4] ✓ 0.4s
</> dx error: 1.7029261167605239e-09
dgamma error: 7.420414216247087e-13
dbeta error: 2.8795057655839487e-12
```

可以看到误差很小，结果正确

3- Batch normalization: alternative backward

由论文公式实现代码如下：


```
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 = np.sum((x - mean) * -0.5 * (var + eps) ** -1.5 * dx_norm, axis = 0)
dm = np.sum(dx_norm * -1 * (var + eps) ** -0.5, axis = 0) + np.sum(dv * (x - mean) * -2 / N, axis = 0)
dx = dx_norm / (var + eps) ** 0.5 + dv * 2 * (x - mean) / N + dm / N
```

最后测试可以发现计算结果是原来的 2 倍多

```
dx difference: 1.2266724949639026e-12
dgamma difference: 0.0
dbeta difference: 0.0
speedup: 2.32x
```

结论分析与体会：

通过本次实验，实现了正则化和批量归一化的概念和方法。在正则化的实验中，学习分别利用 L2 正则化和 dropout 正则化的方法比年过拟合问题；在批量归一化的实验中，通过修改前向传播和后向传播的过程和代码，加入了 gamma 和 beta 即缩放参数和偏移参数，防止深层的神经元出现梯度消失和梯度爆炸问题，导致训练速度过慢的问题，同时还学习了利用 alternative backward，加快了运算速度。

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：

这次实验总体上来说就是内容比较明确，实验过程中代码大多数情况下可以由给定公式推导出来，同时了解了 Batch Normalization，可以使用更大的学习率，训练过程更加稳定，极大提高了训练速度。可以将 bias 置为 0，因为 Batch Normalization 的 Standardization 过程会移除直流分量，所以不再需要 bias。对权重初始化不再敏感，通常权重采样自 0 均值某方差的高斯分布，以往对高斯分布的方差设置十分重要，有了 Batch Normalization 后，对与同一个输出节点相连的权重进行放缩，其标准差 σ 也会放缩同样的倍数，相除抵消。对权重的尺度不再敏感，理由同上，尺度统一由 γ 参数控制，在训练中决定。深层网络可以使用 sigmoid 和 tanh 了，理由同上，BN 抑制了梯度消失。Batch Normalization 具有某种正则作用，不需要太依赖 dropout，减少过拟合。