

## 二、改善神经网络 (Improving Deep Learning Neural Networks)

### Week 1 深度学习的使用层面

课后编程练习 1

训练集, 测试集, 开发集

偏差 (bias) 和方差 (variance)

机器学习基础 (如何根据偏差和方差调整神经网络)

正则化 (逻辑回归和神经网络中的正则化)

关于正则化能够预防过拟合的原因

dropout 正则化方法

对于 dropout 的理解

其他正则化方法

课后编程练习 2

归一化处理

梯度爆炸/消失 (vanishing/exploding of gradients) 问题

针对梯度爆炸/梯度消失进行的权重初始化

梯度检测的数值逼近 (针对梯度的数值逼近)

应用梯度检测

编程练习前需要注意的训练技巧

课后编程练习 3

### Week 2 优化算法

mini-batch 算法

理解 mini-batch

指数加权平均 (exponentially weighted averages, EWA)

对于指数加权平均的理解

偏差修正 (bias correction)

动量梯度下降法 (Momentum gradient descent)

RMSprop (Root Mean Square prop)

Adam (Adaptive Moment Estimation or authorization algorithm 权威算法) 自适应矩阵估计

学习率衰减

局部最优和鞍点

课后编程练习 1

### Week 3 超参数调试、Batch 正则化和程序框架

超参数搜索

超参数搜索范围标尺的选取

关于超参数搜索的一些技巧

Batch 归一化

对神经网络应用 Batch 归一化

Batch 的作用及其原因

测试时期的 Batch 归一化

softmax 回归

关于 softmax 的补充

深度学习框架

tensorflow 的使用

课后编程练习 1

## 二、改善神经网络（Improving Deep Learning Neural Networks）

### Week 1 深度学习的使用层面

#### 课后编程练习 1

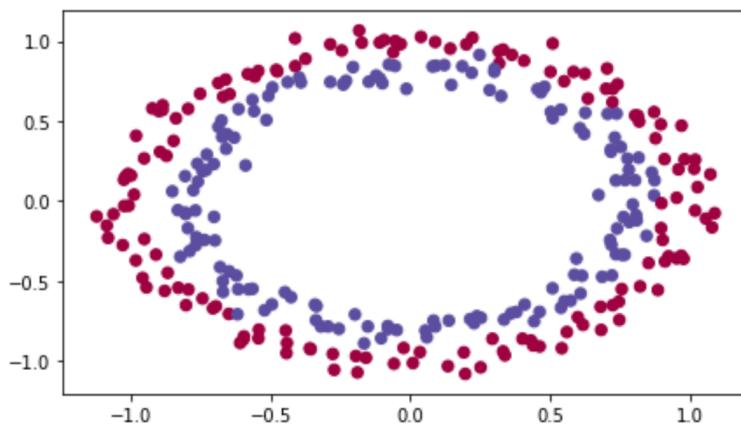
本编程练习无需学习本章内容知识，是有关初始化方法的编程练习。

关于初始化方法，除了之前提到的全零初始化，随机初始化之外，还有一种初始化方法叫 He initialization。即使你知道了前两个方法的效果，也先不要着急，看看前两种初始化方法的可视化效果。

首先看看训练集可视化之后长什么样子：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import sklearn
4 import sklearn.datasets
5 # init_utils.py 文件在文件夹 “Week1_编程练习/编程练习1” 下获取
6 from init_utils import sigmoid, relu, compute_loss, forward_propagation,
backward_propagation
7 from init_utils import update_parameters, predict, load_dataset,
plot_decision_boundary, predict_dec
8
9 %matplotlib inline
10 plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots
11 plt.rcParams['image.interpolation'] = 'nearest'
12 plt.rcParams['image.cmap'] = 'gray'
13
14 # load image dataset: blue/red dots in circles
15 train_X, train_Y, test_X, test_Y = load_dataset()
16 # plt.show() 如果你的运行环境在执行完这段代码后能够显示代码，那么请无视这一行代码
17
18 ##### 注意 #####
19 # 如果提示了有关 scatter c 参数的错误，请尝试使用 np.squeeze() 函数处理传给 c 参数的数据
```

执行后的图片显示如下：



接下来编写神经网络模型，这里作者提供了源代码 init\_utils.py 文件，我们直接引用里面的方法就好，代码如下所示：

```
1 def model(X, Y, learning_rate = 0.01, num_iterations = 15000, print_cost =  
2     True, initialization = "he"):  
3     """  
4         Implements a three-layer neural network: LINEAR->RELU->LINEAR->RELU-  
5         >LINEAR->SIGMOID.  
6  
7     Arguments:  
8         X -- input data, of shape (2, number of examples)  
9         Y -- true "label" vector (containing 0 for red dots; 1 for blue dots),  
10        of shape (1, number of examples)  
11        learning_rate -- learning rate for gradient descent  
12        num_iterations -- number of iterations to run gradient descent  
13        print_cost -- if True, print the cost every 1000 iterations  
14        initialization -- flag to choose which initialization to use  
15        ("zeros", "random" or "he")  
16  
17    Returns:  
18        parameters -- parameters learnt by the model  
19        """  
20  
21        grads = {}  
22        costs = [] # to keep track of the loss  
23        m = X.shape[1] # number of examples  
24        layers_dims = [X.shape[0], 10, 5, 1]  
25  
26        # Initialize parameters dictionary.  
27        if initialization == "zeros":  
28            parameters = initialize_parameters_zeros(layers_dims)  
29        elif initialization == "random":  
30            parameters = initialize_parameters_random(layers_dims)  
31        elif initialization == "he":  
32            parameters = initialize_parameters_he(layers_dims)  
33  
34        # Loop (gradient descent)
```

```

31
32     for i in range(0, num_iterations):
33
34         # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR
35         #> SIGMOID.
36
37         a3, cache = forward_propagation(X, parameters)
38
39         # Loss
40         cost = compute_loss(a3, Y)
41
42         # Backward propagation.
43         grads = backward_propagation(X, Y, cache)
44
45         # Update parameters.
46         parameters = update_parameters(parameters, grads, learning_rate)
47
48         # Print the loss every 1000 iterations
49         if print_cost and i % 1000 == 0:
50             print("Cost after iteration {}: {}".format(i, cost))
51             costs.append(cost)
52
53         # plot the loss
54         plt.plot(costs)
55         plt.ylabel('cost')
56         plt.xlabel('iterations (per hundreds)')
57         plt.title("Learning rate = " + str(learning_rate))
58         plt.show()
59
60     return parameters

```

接下来看看三种初始化的表现：

### 1. 零初始化 (Zero initialization)

```

1 def initialize_parameters_zeros(layers_dims):
2     parameters = {}
3     L = len(layers_dims)                      # number of layers in the network
4
5     for l in range(1, L):
6         ### START CODE HERE ### (~ 2 lines of code)
7         parameters['w' + str(l)] = np.zeros((layers_dims[l],
8                                             layers_dims[l-1]))
9         parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
10        ### END CODE HERE ###
11
12    return parameters

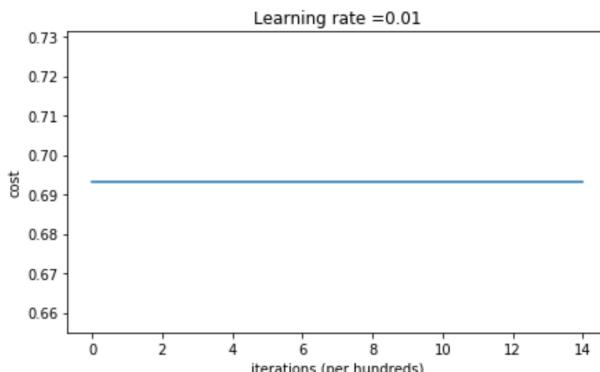
```

训练模型代码：

```
1 parameters = model(train_X, train_Y, initialization = "zeros")
2 print ("On the train set:")
3 predictions_train = predict(train_X, train_Y, parameters)
4 print ("On the test set:")
5 predictions_test = predict(test_X, test_Y, parameters)
```

## 测试结果：

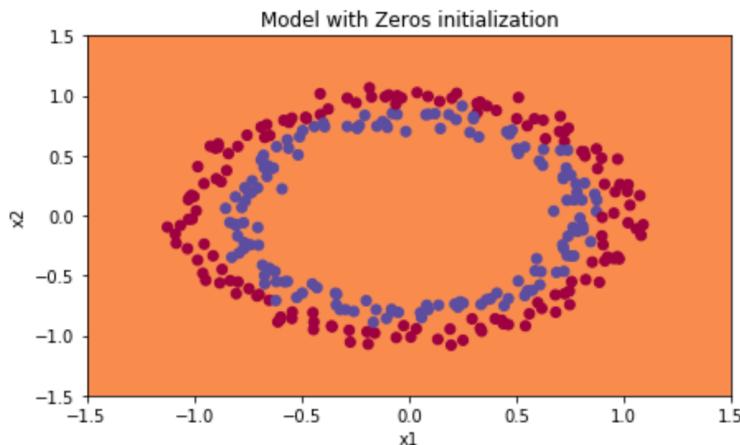
```
Cost after iteration 0: 0.6931471805599453
Cost after iteration 1000: 0.6931471805599453
Cost after iteration 2000: 0.6931471805599453
Cost after iteration 3000: 0.6931471805599453
Cost after iteration 4000: 0.6931471805599453
Cost after iteration 5000: 0.6931471805599453
Cost after iteration 6000: 0.6931471805599453
Cost after iteration 7000: 0.6931471805599453
Cost after iteration 8000: 0.6931471805599453
Cost after iteration 9000: 0.6931471805599453
Cost after iteration 10000: 0.6931471805599455
Cost after iteration 11000: 0.6931471805599453
Cost after iteration 12000: 0.6931471805599453
Cost after iteration 13000: 0.6931471805599453
Cost after iteration 14000: 0.6931471805599453
```



```
On the train set:  
Accuracy: 0.5  
On the test set:  
Accuracy: 0.5
```

```
1 print ("predictions_train = " + str(predictions_train))  
2 print ("predictions test = " + str(predictions test))
```

```
1 plt.title("Model with Zeros initialization")
2 axes = plt.gca()
3 axes.set_xlim([-1.5,1.5])
4 axes.set_ylim([-1.5,1.5])
5 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_x,
train_y)
```



由于对称性的原因导致最终结果并不理想，很烂。

## 2. 随机初始化 (Random initialization)

## 直接上代码！！

```
1 def initialize_parameters_random(layers_dims):
2     np.random.seed(3)                      # This seed makes sure your "random"
3     numbers will be the as ours
4     parameters = {}
5     L = len(layers_dims)                  # integer representing the number of
6     layers
7
8     for l in range(1, L):
9         ### START CODE HERE ### (~ 2 lines of code)
10        parameters['W' + str(l)] = np.random.randn(layers_dims[l],
11                                         layers_dims[l-1]) * 10
12        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
13        ### END CODE HERE ###
14
15    return parameters
```

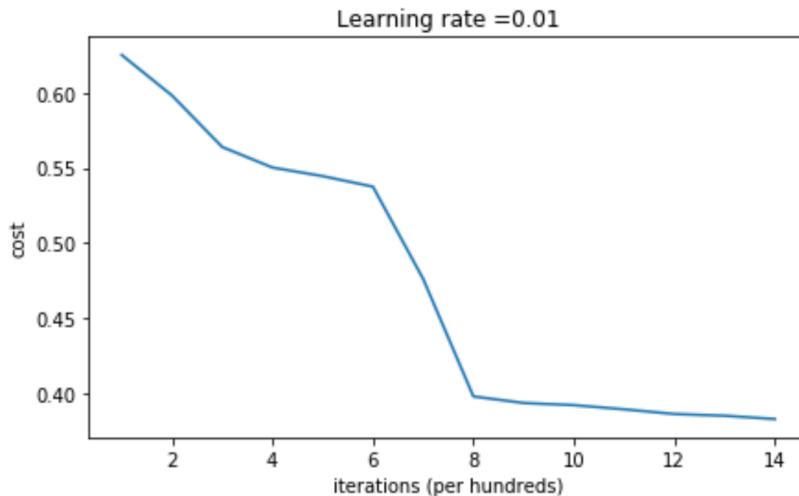
训练模型以及测试模型：

```
1 parameters = model(train_X, train_Y, initialization = "random")
2 print ("On the train set:")
3 predictions_train = predict(train_X, train_Y, parameters)
4 print ("On the test set:")
5 predictions_test = predict(test_X, test_Y, parameters)
```

```

Cost after iteration 0: inf
Cost after iteration 1000: 0.6250884962121392
Cost after iteration 2000: 0.5981371467489438
Cost after iteration 3000: 0.5638539771863162
Cost after iteration 4000: 0.5501704762630747
Cost after iteration 5000: 0.5444592806792145
Cost after iteration 6000: 0.5374509252365552
Cost after iteration 7000: 0.4760640415643904
Cost after iteration 8000: 0.3978146724300182
Cost after iteration 9000: 0.3934785833165248
Cost after iteration 10000: 0.3920322287285902
Cost after iteration 11000: 0.38924754816043866
Cost after iteration 12000: 0.38615976417756703
Cost after iteration 13000: 0.38498687252939306
Cost after iteration 14000: 0.38278602219555746

```



On the train set:

Accuracy: 0.83

On the test set:

Accuracy: 0.86

这里 Inf 的原因是我们在初始化的参数太大了，导致文件中内置的 sigmoid 函数里面的 np.exp() 函数溢出了，如果不想溢出的话，可以将这里更换成 `s = scipy.special.expit(x)` .

```

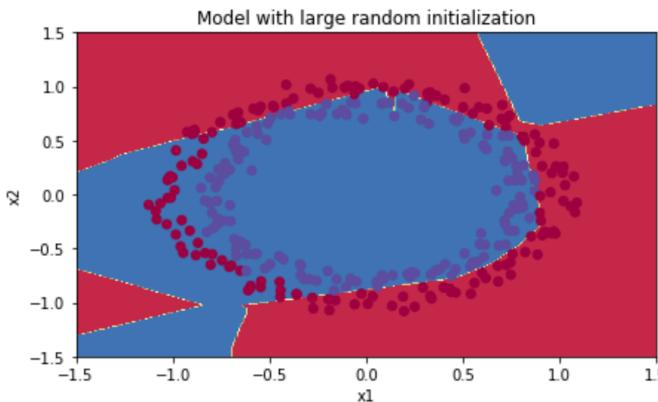
1 print (predictions_train)
2 print (predictions_test)
3
4 output:
5 [[1 0 1 1 0 0 1 1 1 1 0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0 1
6   1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 0
7   0 0 0 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 1 0
8   1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0
9   0 0 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1
10  1 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 0 1
11  0 1 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0 1 1
12  1 1 0 1 1 0 1 1 1 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 1
13  1 1 1 1 0 0 0 1 1 1 1 0 ]]

14 [[1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1
15  0 1 1 0 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 1 0
16  1 1 1 1 1 0 1 0 0 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 0 1]
```

```

1 plt.title("Model with large random initialization")
2 axes = plt.gca()
3 axes.set_xlim([-1.5,1.5])
4 axes.set_ylim([-1.5,1.5])
5 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
train_Y)

```



### 3. He initialization

至于这个翻译成“他初始化”还是其他术语，先暂且不提。如果你仔细看过上一章编程作业4中，作者自己实现的内置函数的话，你会发现初始化函数对 $w$ 的初始化代码为`parameters["W" + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) / np.sqrt(layer_dims[l-1])`，我认为就是从这里获得的灵感。其实简单来说，He initialization 在`parameters["W" + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])`之后乘上一个`np.sqrt(2./layer_dims[l-1])`，整体代码如下：

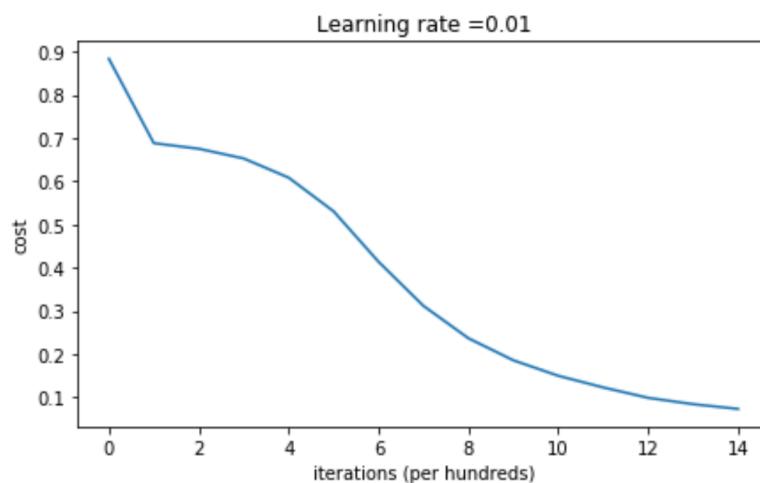
```

1 def initialize_parameters_he(layers_dims):
2     np.random.seed(3)
3     parameters = {}
4     L = len(layers_dims) - 1 # integer representing the number of layers
5
6     for l in range(1, L + 1):
7         ##### START CODE HERE ##### (~ 2 lines of code)
8         parameters['W' + str(l)] = np.random.randn(layers_dims[l],
layers_dims[l-1]) * np.sqrt(2./layers_dims[l-1])
9         parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
10        ##### END CODE HERE #####
11
12    return parameters

```

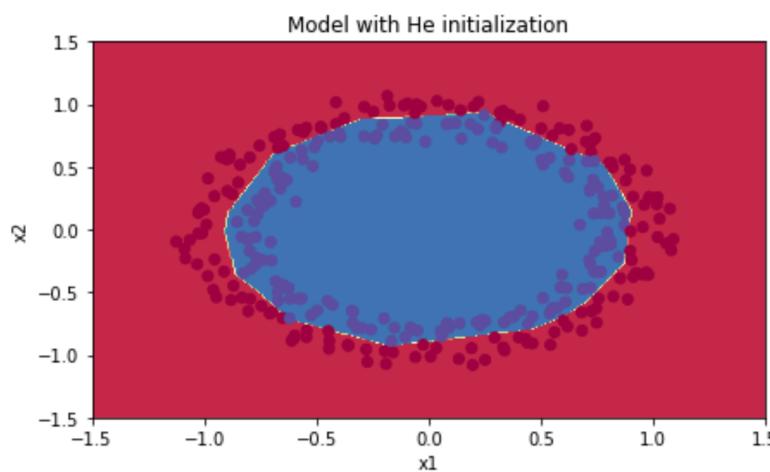
这个方法的表现是非常优秀的，训练结果如下图所示：

```
Cost after iteration 0: 0.8830537463419761
Cost after iteration 1000: 0.6879825919728063
Cost after iteration 2000: 0.6751286264523371
Cost after iteration 3000: 0.6526117768893807
Cost after iteration 4000: 0.6082958970572938
Cost after iteration 5000: 0.5304944491717495
Cost after iteration 6000: 0.4138645817071795
Cost after iteration 7000: 0.3117803464844414
Cost after iteration 8000: 0.2369621533032257
Cost after iteration 9000: 0.18597287209206845
Cost after iteration 10000: 0.1501555628037181
Cost after iteration 11000: 0.12325079292273548
Cost after iteration 12000: 0.09917746546525937
Cost after iteration 13000: 0.08457055954024273
Cost after iteration 14000: 0.07357895962677366
```



```
On the train set:
Accuracy: 0.9933333333333333
On the test set:
Accuracy: 0.96
```

决策边界图如下所示：



那么这个 He initialization 究竟应该怎么读呢？我搜了一下，这个方法的提出叫 Kaiming He，中文名何恺明，个人经历十分“开挂”，感兴趣可以去搜搜看，我个人认为这个方法应该叫“何氏初始化”。

训练集，测试集，开发集

超参数（神经网络层数，每层的节点数，学习率，激活函数的选择等等）的确定，对于一个神经网络的质量起到关键作用。但即使是行业大佬也很难一次给出最优参数。因此通过多次对模型的训练迭代来选取超参数。然而不论是大型还是小型的神经网络，训练模型需要时间来获得反馈，因此对于训练集的选取尤为重要。

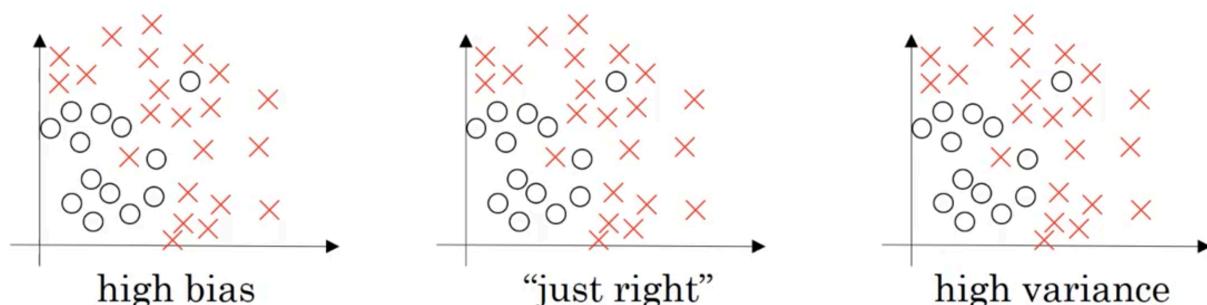
一般把整个数据集划分为三种：训练集，验证集（development set）/交叉验证集（cross validation set）以及测试集。训练集用于训练神经网络，验证集则用于验证算法模型，确定合适的算法模型，测试集用于测试算法模型的结果，用于无偏估计。若算法模型不被要求是无偏的，则可以省略测试集合。具体区别可参考这里：[https://blog.csdn.net/weixin\\_42029738/article/details/81066640](https://blog.csdn.net/weixin_42029738/article/details/81066640)。对于数据集的选择需要符合使用场景，例如教程中所举的例子，若要为某个手机应用程序部署神经网络，来识别用户上传的图片是否是喵，那么使用网路上的图片训练神经网络是不太合适的。因为网络中的图片有很多高分辨率，专业摄影的图片，然而这个神经网络最终是要处理用户上传的，低分辨率的，非专业拍摄的图像，这使得验证集和测试集不属于同一分布，导致神经网络的质量不佳。

对于训练集，验证集以及测试集的比例，在数据量较少时（教程建议为百万数据集以下），建议  $6:2:2$ ，但在数据量较大时，可采用  $98:1:1$  或  $99.5:0.25:0.25$ 。

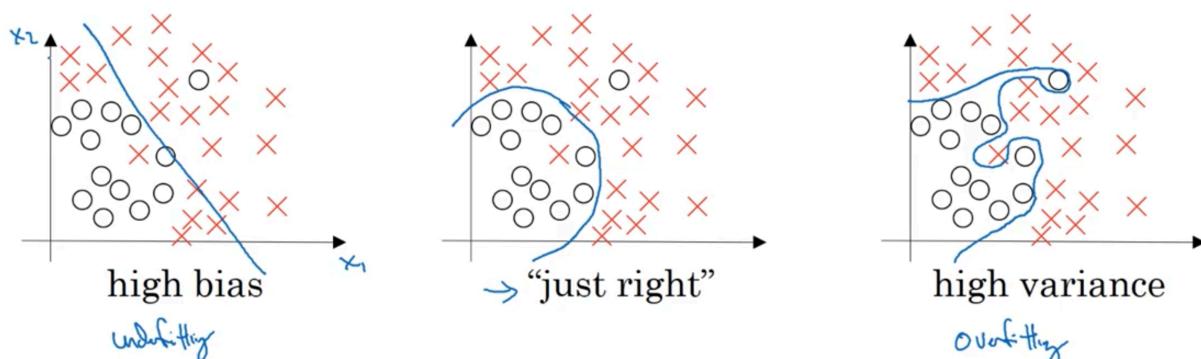
### 偏差（bias）和方差（variance）

偏差和方差是衡量神经网络质量的两个指标，那么接下来就来看看什么是偏差和方差。

有下图的二维特征向量的数据集：



若拟合方式从左到右分别为直线，合适的曲线和过于灵活的曲线，则情况如下图所示：



可以直观的看出，直线并不能很好的拟合数据，导致预测数据有明显偏差，被称为欠拟合（underfitting），特征是高偏差（high bias）；过度灵活的曲线完全拟合给定的数据集，被称为过拟合（overfitting），特征为高方差（high variance）。

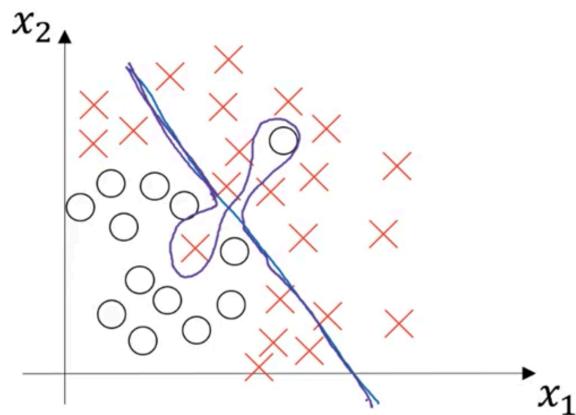
上述情况为特征向量是二维的情况，对于高维特征向量很难直观地像上述图例那样看出，这类数据集需要根据对训练集和验证集的处理结果看出。在最优误差（又称为基本误差，可以理解为人眼对于喵脸识别的误差至于模型对于喵脸识别误差的关系，我理解为是被训练模型误差的参照标准）较小（接近1%），且训练集和验证集同分布的情况下，规律如下：

1. 若训练集误差为 1%，验证集误差为 11%，则称为过拟合，高方差。
2. 若训练集误差为 15%，验证集误差为 16%，则称为欠拟合，高偏差。

3. 若训练集误差为 15%，验证集误差为 30%，则称为高方差，高偏差。

4. 若训练集误差为 0.5%，验证集误差为 1%，则称为低方差，低偏差。

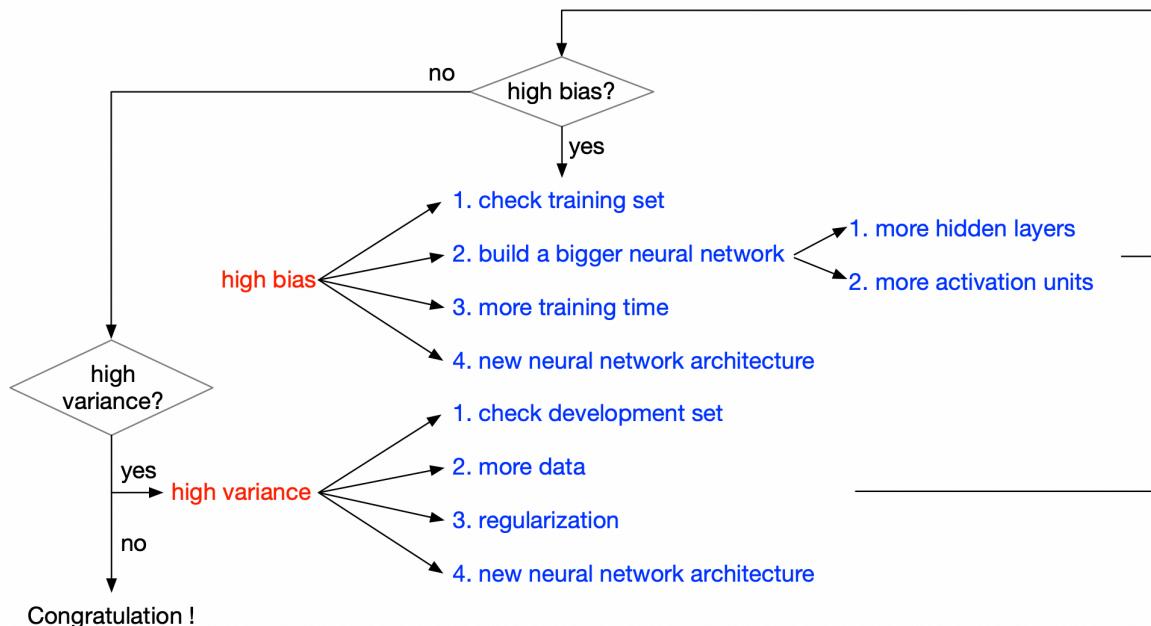
对于上述规律中的第三种情况，可以用二维特征向量数据集表现为紫色的曲线：



这条曲线既有直线拟合欠拟合的特征，又有过度灵活曲线过拟合的特征。

### 机器学习基础（如何根据偏差和方差调整神经网络）

之前学习的内容告诉我们，我们最终的目标是构建一个具有合适偏差方差的神经网络。一般调整神经网络和数据集，按照下列步骤进行：



上图中红色文字表示模型处理完训练集 (training set) 或验证集 (development set) 之后发现的问题，蓝色文字表示解决办法。菱形框表示是否存在框内内容问题。

值得注意的是，教程中提出，在机器学习时代，在调整某个参数时很难保证另一个参数不发生变化（或发生微小的变化）。例如降低偏差值，会使得方差值也被改变。但在深度学习时代，则不必担心这个问题，因为有工具使得偏差值降低时，仅微弱的改变方差值，反之亦然。例如，若我想要降低偏差值而对方差值不造成太大影响，那么在适当正则化 (regularization) 的情况下，构建规模更大的神经网络即可。若我想要降低方差值而不对偏差值造成较大影响，则增加数据集即可。下一节会讨论正则化的问题。

### 正则化（逻辑回归和神经网络中的正则化）

在解决高方差问题的方法中，增加数据量和对参数进行正则化被认为是十分有效的。由于数据有时候很难被获取，或者获取的成本很高，因此正则化是较为常用的方法。

逻辑回归的正则化根据范数种类分为两种： $L_1$  范数以及  $L_2$  范数。我们一般对参数  $w$  进行正则化，原因可以总结为参数  $w$  对模型处理数据的结果影响较参数  $b$  更明显，详细可参阅以下博客：

1. [为什么一般不对偏置b进行正则化？ -- CSDN](#)
2. [神经网络一般不对偏置项b进行正则化的原因 -- CSDN](#)

$L_1$  和  $L_2$  正规化的定义如下：

$$\begin{aligned} L_1 \text{ regularization : } J(w, b) &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} \sum_{j=1}^m |w_j| \\ &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} \| w \|_1 \end{aligned}$$

$$\begin{aligned} L_2 \text{ regularization : } J(w, b) &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} \sum_{j=1}^m w_j^2 \\ &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} w^T w \\ &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} \| w \|_2^2 \end{aligned}$$

公式中的  $\lambda$  表示正规化参数，由验证集和交叉验证集来确定，在后面的编程作业中，该参数被赋予变量名 `lambd` 用于区别 python 的内置关键字 `lambda`。两种正则化方法，一般选用  $L_2$ 。因为  $L_1$  会使得  $w$  向量变稀疏（就是向量中的数据有很多 0），有人认为这有利于压缩模型（由于 0 很多，可以节省内存），但实际上并没有节省多少。因此我们选用  $L_2$  正则化方法。

在神经网络的前向传播中， $L_2$  正则化的  $J(w, b)$  公式如下：

$$\begin{aligned} L_2 \text{ regulation : } J(W, b) &= \frac{1}{m} \sum_{j=1}^m L(\hat{y}, y) + \frac{\lambda}{2m} \sum_{l=1}^L \| W^{[l]} \|^2 \\ \| W^{[l]} \|^2 &= \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \end{aligned}$$

上面公式中的  $W^{[l]}$  维度为  $(n^{[l-1]}, n^{[l]})$ ，这里是未转置的向量叠加而成的矩阵，可对比第一章第四周第三节课课件中对  $W^{[l]}$  维度的定义。关于  $\| W^{[l]} \|^2$  没有下标的原因是， $L_2$  范数的叫法只适用于向量，虽然计算方法并无差别，但是对于矩阵来说，这个范数叫做 Frobenius 范式，这里的原因参考自 [Frobenius范数的定义--知乎](#)。

由于前向传播中  $J(W, b)$  函数的变化，导致反向传播中对参数  $W$  的导数变为  $dW^{[l]} = (\text{origin part}) + \frac{\lambda}{m} * W^{[l]}$ ，那么对于参数  $W^{[l]}$  的更新，公式中的  $dW^{[l]}$  替换为上述公式。我们对这个公式做一个变形，过程如下：

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha * ((\text{origin part}) + \frac{\lambda}{m} * W^{[l]}) \\ &= (1 - \alpha * \frac{\lambda}{m}) W^{[l]} - \alpha (\text{origin part}) \end{aligned}$$

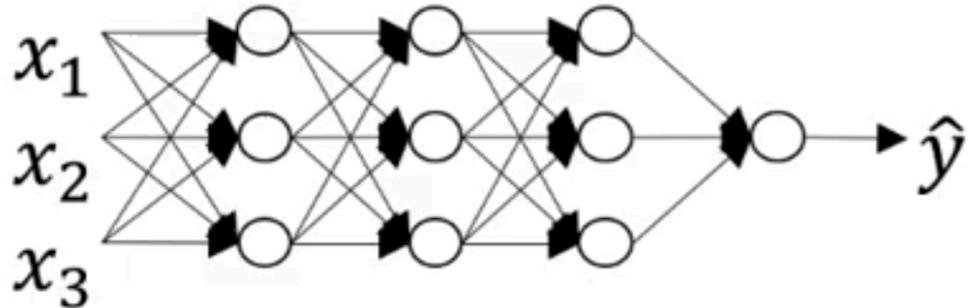
我们可以看到新的公式给参数（权重） $W^{[l]}$  乘以了一个小于 1 的系数，这使得参数值比原先要小，因此  $L_2$  范数正则化（范数正则化的原因可能是参与计算的是向量，而非矩阵）又被称为权重衰减（weight decay）。

上述的内容只是阐述了对正则化参数的计算过程和参数定义，那么正则化能够预防过拟合的原因，在下一节阐述。

### 关于正则化能够预防过拟合的原因

关于这个原因，教程中给出了两种较为直观的解释：

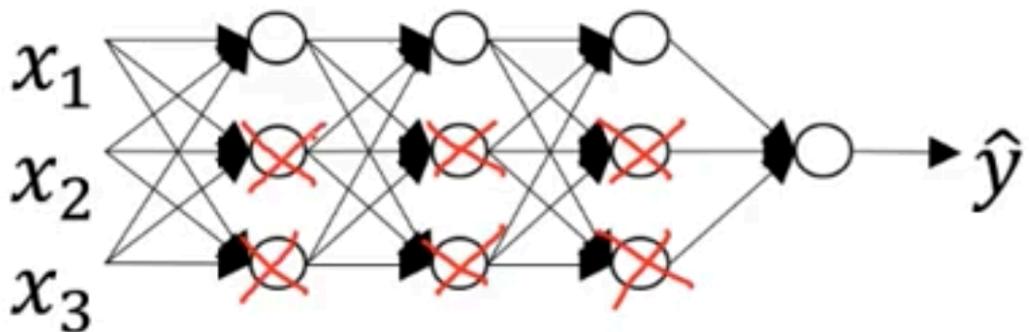
1. 给定一个深度神经网络如下图所示（可能不够深度，但仅做示意）：



上节内容提到，对于更新后的  $w$  更新函数如下所示：

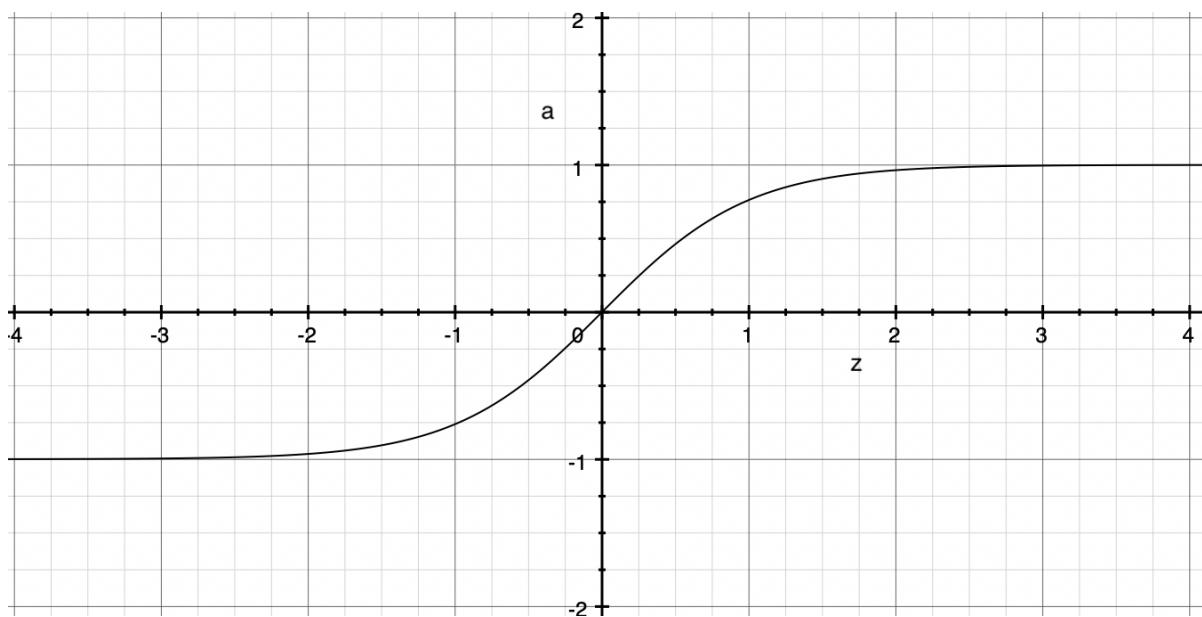
$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha * ((\text{origin part}) + \frac{\lambda}{m} * W^{[l]}) \\ &= (1 - \alpha * \frac{\lambda}{m}) W^{[l]} - \alpha (\text{origin part}) \end{aligned}$$

当  $\lambda$  越大，则更新后的  $W^{[l]}$  越小。这里教程中对“当  $\lambda$  越大，则更新后的  $W^{[l]}$  越小”的解释为：损失函数中添加了正则化项，当  $\lambda$  越大，使得参数  $w$  的值越小，个人认为是因为参数更新函数中的  $\lambda$  最终决定了  $w$  的大小，如有错误请指出。如果  $\lambda$  的取值足够大以至于  $w \approx 0$ ，那么这将使得多数激活单元几乎失去作用（仅仅剩下  $b$  参数，影响很小），那么可能会使得上图的神经网络变成如下情况：



很直观的可以看出，之前的深度神经网络几乎成了只有深度的逻辑回归，这将使得神经网络的效果与欠拟合的曲线相似，若  $\lambda$  非常小，那么情况恰恰相反，因此，在这个范围内可以选取合适的  $lambda$  值。

2. 另一种比较直观的解释为，激活函数  $a = \tanh(z)$  的图像如下所示：



当  $\lambda$  非常大时， $|z|$  的值非常小， $z$  在 0 附近的范围就会很小，这使得这个范围内的函数近似与线性函数。前一章的内容也提到过，当神经网络激活单元的激活函数为线性函数时，神经网络整体效果跟逻辑回归效果是一样的，由此可见，当  $\lambda$  非常大时，神经网络近似于逻辑回归的效果，也就避免了过拟合的问题。

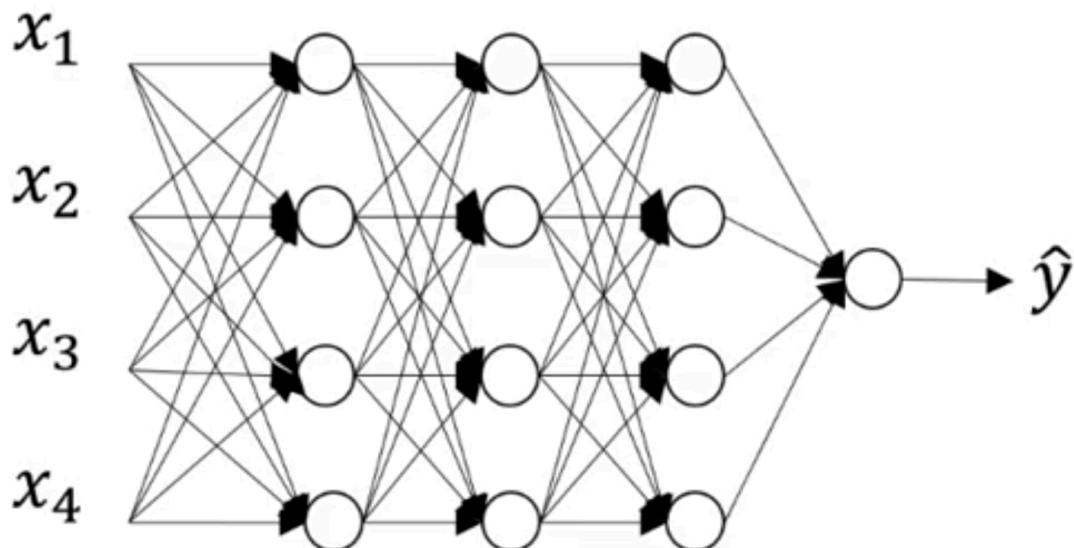
教程中还指出，当调试梯度下降函数时，使用的就是新的（含有正则化项的）代价函数，这时代价函数对于梯度下降的每隔调幅都单调递减。我对这里不是很理解，百度谷歌了调幅，但是没有找到有关代价函数调幅的知识，教程后面也许会提到吧，这里不做解释（我也不知道QAQ）。

下个内容会介绍一个叫 dropout 的正则化方法。

### dropout 正则化方法

首先，不得不说一下，网络上流传的网易云课堂的视频资源，在本节视频内容的中文字幕错误比较严重，尤其是视频的后半部分，这里我是根据视频的英文字幕理解学习的，如有错误的地方请指出。

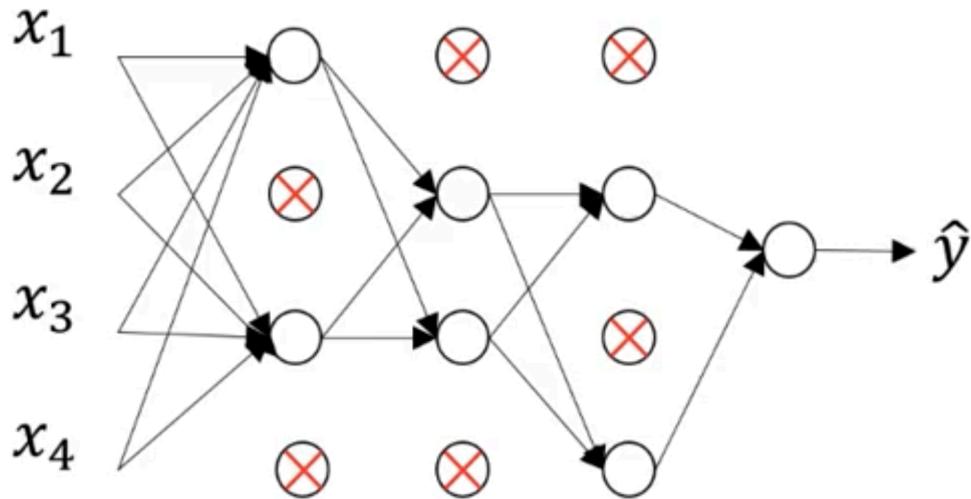
本节内容讲述了 dropout 算法的步骤。给定一个深度神经网络模型如下图



假定该深度神经网络模型存在过拟合问题，dropout 的想法是，通过随机去除深度神经网络模型中每层的部分节点的方式，避免过拟合（这种方式奏效的原因，我个人理解为，可以参考  $L_2$  正则化方法在  $\lambda$  非常大时，会使得部分  $w$  参数几乎为 0，这使得部分节点几乎失效，模型的处理效果与逻辑回归相似。）。这个方法的大致步骤如下：

1. 遍历神经网络中的每一个隐藏层
2. 对每个隐藏层的每个节点抛硬币决定是否去除这个节点
3. 删除进入被去除节点，以及从被去除节点出发的箭头

进行完上述操作之后，得到的模型假定如下图所示：



上述做法被称作是 dropout。下面来看看如何用代码实现 dropout，以及其他的一些细节。

对于某个神经网络的  $l$  层隐藏层向量，这里记做  $a^{[l]}$ 。若想按照一定概率去除  $a^{[l]}$  中的节点，可以生成一个与  $a^{[l]}$  维度相同的随机向量，按照随机向量中数据是否大于某个概率，来判断是否去除  $a^{[l]}$  中的某个节点。上述描述有点绕，但是代码非常直观，如下段代码所示：

```

1 | # 已知 al
2 | keep_prob = 0.8 # 设定去除某个节点的概率为 80%
3 | dl = np.random.randn(al.shape[0], al.shape[1]) > keep_prob
4 | al = al * dl

```

上段代码中的  $dl$  最终为一个仅包含 bool 值数据的向量，但在 python 中，在做运算时，会将 bool 值的 False 和 True 分别转换成 0 和 1。但除了上述操作之外，我们还需要一行代码

```

1 | al = al / keep_prob

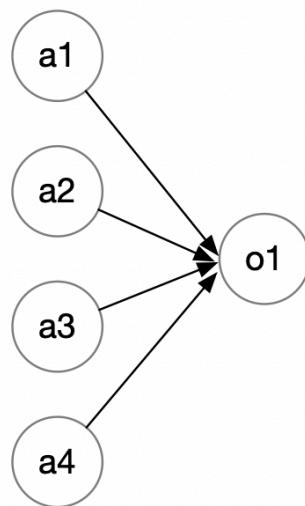
```

这是因为我们需要让  $al$  本身的期望值不变。这一步操作也被称为 inverse dropout，它使得测试阶段的 scaling problems 减少。在早期的 dropout 操作中，是没有这一步操作的，这会使得测试阶段的平均值变得越来越复杂（这里的平均值个人理解为是损失函数的平均值  $\$1/m\$$ ）。接下来的视频内容中，我没有理解中文字幕，在这里贴出原文：you'll notice that for different training examples, you zero out different hidden units. In fact, if you make multiple passes through the same training set, then on different passes through the training set, you should randomly zero out different hidden units. 我个人理解这段话是在说，对同一个数据集做多次 dropout 操作时，每一次去除的隐藏层单元与也不相同，这不论是在前向传播还是反向传播中都是一样的。

在测试阶段，我们需要获得一个确定的，关于测试集的预测值，因此我们在测试阶段不使用 dropout。之前的 dropout inverse 操作也是为了使得测试阶段的  $a^{[l]}$  期望值不变。dropout 能够起到作用的具体原因将在下一块儿内容进行阐述。

**对于 dropout 的理解**

这节内容来回答为什么 dropout 能起作用。首先，根据上节内容，我们可以发现，dropout 的直观效果就是将一个隐藏层节点较多的大的神经网络，变成一个隐藏层节点较少的，小的神经网络，每次迭代都会使得神经网络变小（每次迭代都会随机删减一些节点，但每次都基于整个节点，而不是上次迭代保留下来的节点）。另一种直观的解释是：dropout 会导致权重（参数  $w$ ）的平方范数收缩。推导过程为：假设有一个单个神经元（single unit）如下图：



通过 dropout， $a_1$  至  $a_4$  中会有单元被清除，因为这个原因，输出单元  $o_1$  不能够依赖于任何一个输入单元（我个人理解为，是 dropout 这个算法使得  $o_1$  不能够依赖输入单元，因为它们会被算法随机清除掉，即使存在权重值很大的单元，最后也有机会被清除掉），由此产生类似于  $L_2$  一样的，对权重的平方范数进行收缩的效果（这里我个人理解为， $a_1$  至  $a_4$  中某个单元被去除后，影响到  $\parallel w \parallel^2$  的计算，使其变小，但这将在更新参数的过程中，使得参数变大，除非它是对下一次的范数计算进行影响，所以对我而言，这样理解不太直观，我个人还是偏向于清除单元那个解释方式。）。 $L_2$  正则化对不同权重有着不同效果的衰减，这取决于倍增的激活函数的大小（原文是：depending on the size of activations being multiplied that way 没明白啥叫个倍增的激活函数，如果跟不同权重有关系的话，我觉得与  $z = wx + b$  有关系）。由于 dropout 随机去除节点的特性，所以它能够适应于不同的输入范围。

在实现 dropout 的时，可以根据每个隐藏层的维度去确定 `keep_prob`，若隐藏层的比较大，则可以调低 `kepp_prob` 的数值，但是对于最终输出层（一般为  $(1,1)$ ）可以设置为 `keep_prob = 1`，即不进行 dropout。

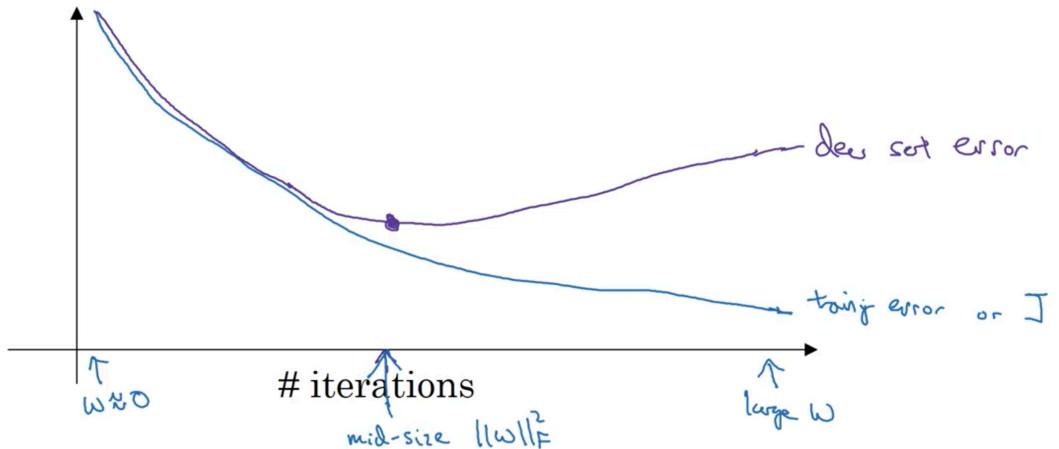
dropout 在计算机视觉上常用，因为数据集会比较少，容易过拟合。但是 dropout 的缺点在于代价函数不明确，难以 debug。

## 其他正则化方法

解决过拟合问题的另一种办法就是扩大数据集。对于喵脸识别来说，当喵咪的数据集太小时，可以适当对图片进行景象和裁剪操作，但要保证处理后的图像还能被人看出是个喵来（例如不推荐上下颠倒，反色等处理）。但是要注意，这样处理后的数据集容易造成过拟合，因为尽管是对原数据集进行了一番改动，它还是不如新的数据集那样，能够使得模型学到更多特征，因此需要 dropout 正则化方法预防过拟合问题。另一个例子就是做数字识别时，可以适当的对数字进行扭曲处理，如下图所示：



接下来进入正题，除了之前提到的  $L_2$  和 dropout 正则化方法之外，还有一种方法叫做 early stop。我们在使用 plt 绘制成本函数  $J$  每次迭代后值的曲线时，也可以同时绘制模型对验证集处理的误差（例如：分类误差，逻辑误差，代价函数，对数误差），可能会得到如下曲线：



Andrew Ng

蓝色曲线代表损失函数每次迭代后的数值，紫色曲线代表验证集误差。直观的来看，为了减少误差，我们应该选择中间那个 W 值，使得 Frobenius 范数较小，这与  $L_2$  正则化方法类似。相比于  $L_2$  正则化方法，这种方法能够很方便的找到 w 参数，然而它也有缺点。我们展开来说。

在对损失函数的优化上，我们可以通过梯度下降的方法，但是除此之外，还有其他方法，例如 Adam 等等。在对高偏差（欠拟合）或高方差（过拟合）上，我们各自有一套方法去处理它们，之前也提到过。那么我想说的是，一段时间处理一个问题，这种方法是有必要的，它能够使得被处理的各个问题之间互不干扰，这个思路叫做正交化（Orthogonalization），后面还会提及。仔细一想就会发现，early stop 的问题在于，我们确实可以在选择中间那个 w 时，获得最低的验证集误差，但同时会影响到成本函数的优化，因为此时成本函数并非是最低值，但你停止了对成本函数 J 的优化，与此同时，你还想要不发生过拟合的问题，这会使得对参数调整变得复杂。而  $L_2$  正则化方法不会存在这个问题，它使得超参数在分解搜索空间上变得容易，并且更容易搜索（原话为：It makes search space of hyper parameters easier to decompose and easier to search over，我不是很理解这里的意思）。但是缺点是你需要不断的尝试  $\lambda$  的值，这会使得计算成本很大，这也是为什么我们之前说：相比于  $L_2$  正则化方法，这种方法能够很方便的找到 w 参数。

## 课后编程练习 2

这个编程练习想让你体会到正则化的作用。首先它给出了一个问题，法国足球公司想让你决定守门员应该朝球场的什么位置踢足球，能让他们团队的球员用头顶到足球。我们按照练习中的顺序一步一步来。

这个公司给定了你数据集（代码中的数据集可以在文件夹“Week1 / 2/datasets”

教程中也给了帮助函数和测试函数（这些函数文件可以在文件夹“Week1\_编程练习/编程练习 2/functions”下找到）

引用的头文件和其他文件如下：

```

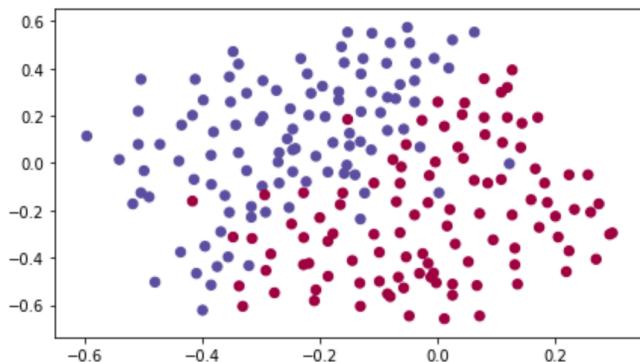
1 # import packages
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from reg_utils import sigmoid, relu, plot_decision_boundary,
5     initialize_parameters, load_2D_dataset, predict_dec
6 from reg_utils import compute_cost, predict, forward_propagation,
7     backward_propagation, update_parameters
8 import sklearn
9 import sklearn.datasets
10 import scipy.io
11 from testCases import *
12
13 plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots
14 plt.rcParams['image.interpolation'] = 'nearest'
15 plt.rcParams['image.cmap'] = 'gray'

```

测试没有问题后，获取训练集数据和测试集数据

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

运行后如下图所示



每个点对应于足球场上的位置，在该位置上，法国守门员从足球场左侧射出球后，足球运动员用他/她的头部击中了球。蓝点表示法国队员用头击中球的位置，红点表示其他国家队队员用头击中球的位置。你的目标是找到法国队守门员的击球区域。看起来数据有些杂乱，但是大概能找出一条直线分开上半部分的蓝色和下半部分的红色。

我们先用非正则化的模型，然后学着去正则化这个模型，再决定是用什么正则化模型去解决这个问题。非正则化模型代码如下：

```

1 def model(X, Y, learning_rate = 0.3, num_iterations = 30000, print_cost =
2     True, lambd = 0, keep_prob = 1):
3     grads = {}
4     costs = [] # to keep track of the cost
5     m = X.shape[1] # number of examples
6     layers_dims = [X.shape[0], 20, 3, 1]
7
8     # Initialize parameters dictionary.
9     parameters = initialize_parameters(layers_dims)

```

```

10     # Loop (gradient descent)
11
12     for i in range(0, num_iterations):
13
14         # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR
15         # -> SIGMOID.
16
17         if keep_prob == 1:
18             a3, cache = forward_propagation(X, parameters)
19         elif keep_prob < 1:
20             a3, cache = forward_propagation_with_dropout(X, parameters,
21
22             keep_prob)
23
24         # Cost function
25         if lambd == 0:
26             cost = compute_cost(a3, Y)
27         else:
28             cost = compute_cost_with_regularization(a3, Y, parameters,
29             lambd)
30
31         # Backward propagation.
32         assert(lambd==0 or keep_prob==1)      # it is possible to use both
33         L2 regularization and dropout,
34                                         # but this assignment will
35         only explore one at a time
36         if lambd == 0 and keep_prob == 1:
37             grads = backward_propagation(X, Y, cache)
38         elif lambd != 0:
39             grads = backward_propagation_with_regularization(X, Y, cache,
40             lambd)
41         elif keep_prob < 1:
42             grads = backward_propagation_with_dropout(X, Y, cache,
43
44             keep_prob)
45
46         # Update parameters.
47         parameters = update_parameters(parameters, grads, learning_rate)
48
49         # Print the loss every 10000 iterations
50         if print_cost and i % 10000 == 0:
51             print("Cost after iteration {}: {}".format(i, cost))
52         if print_cost and i % 1000 == 0:
53             costs.append(cost)
54
55         # plot the cost
56         plt.plot(costs)
57         plt.ylabel('cost')
58         plt.xlabel('iterations (x1,000)')
59         plt.title("Learning rate =" + str(learning_rate))
60         plt.show()

```

52

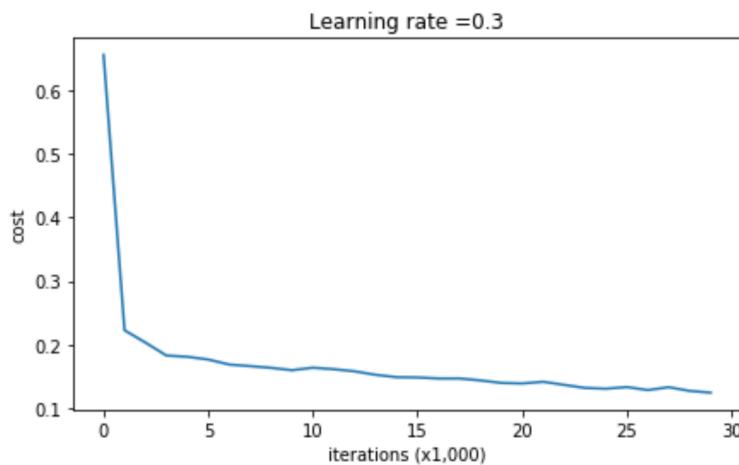
```
    return parameters
```

接着执行下述代码查看这个模型的效果：

```
1 parameters = model(train_X, train_Y)
2 print ("On the training set:")
3 predictions_train = predict(train_X, train_Y, parameters)
4 print ("On the test set:")
5 predictions_test = predict(test_X, test_Y, parameters)
```

执行效果如下图所示

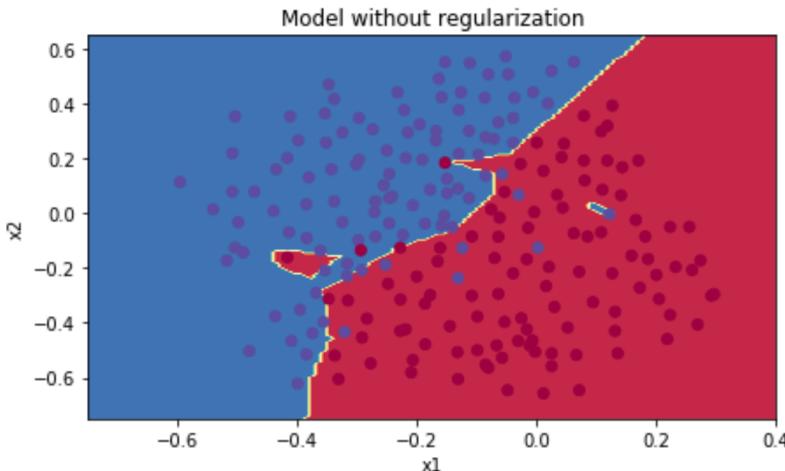
```
Cost after iteration 0: 0.6557412523481002
Cost after iteration 10000: 0.16329987525724213
Cost after iteration 20000: 0.1385164242326018
```



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

我们在检查一下决策边界

```
1 plt.title("Model without regularization")
2 axes = plt.gca()
3 axes.set_xlim([-0.75,0.40])
4 axes.set_ylim([-0.75,0.65])
5 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
train_Y)
```



可以发现，这个模型将噪点也做了分离，并且训练集和测试集之间的准确率相差了 3%，表现出了过耦合的特征。接下来我们看看解决过耦合的两种正则化方法。

首先是 L2 正则化方法，他将代价函数修改为如下形式：

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

接下来就是去实现 `compute_cost_with_regularization()` 这个函数，实现方法如下段代码所示：

```

1 def compute_cost_with_regularization(A3, Y, parameters, lambd):
2     m = Y.shape[1]
3     W1 = parameters["W1"]
4     W2 = parameters["W2"]
5     W3 = parameters["W3"]
6
7     cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-
8     entropy part of the cost
9
10    ### START CODE HERE ### (approx. 1 line)
11    L2_regularization_cost = (lambd / (2*m)) * (np.sum(np.square(W1)) +
12        np.sum(np.square(W2)) + np.sum(np.square(W3)))
13    ### END CODE HERE ###
14    cost = cross_entropy_cost + L2_regularization_cost
15
16    return cost

```

当然啦，除了前向传播的代价函数需要改变，反向传播的代价函数也需要做出改变，代码如下：

```

1 def backward_propagation_with_regularization(X, Y, cache, lambd):
2     m = X.shape[1]
3     (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache
4
5     dZ3 = A3 - Y
6
7     ### START CODE HERE ### (approx. 1 line)
8     dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd/m) * W3
9     ### END CODE HERE ###

```

```

10 db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
11
12 dA2 = np.dot(W3.T, dZ3)
13 dZ2 = np.multiply(dA2, np.int64(A2 > 0))
14 ### START CODE HERE ### (approx. 1 line)
15 dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd/m) * W2
16 ### END CODE HERE ###
17 db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
18
19 dA1 = np.dot(W2.T, dZ2)
20 dZ1 = np.multiply(dA1, np.int64(A1 > 0))
21 ### START CODE HERE ### (approx. 1 line)
22 dW1 = 1./m * np.dot(dZ1, X.T) + (lambd/m) * W1
23 ### END CODE HERE ###
24 db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
25
26 gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
27             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
28             "dZ1": dZ1, "dW1": dW1, "db1": db1}
29
return gradients

```

那么我们测试一下使用了  $L_2$  正则化方法的模型，测试代码和结果如下

```

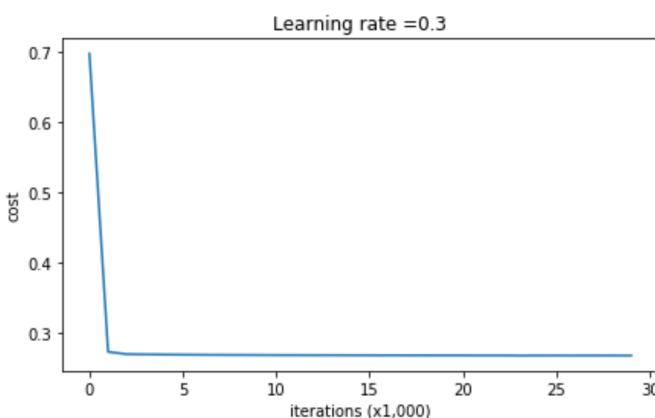
1 parameters = model(train_X, train_Y, lambd = 0.7)
2 print ("On the train set:")
3 predictions_train = predict(train_X, train_Y, parameters)
4 print ("On the test set:")
5 predictions_test = predict(test_X, test_Y, parameters)

```

```

Cost after iteration 0: 0.6974484493131264
Cost after iteration 10000: 0.2684918873282238
Cost after iteration 20000: 0.26809163371273004

```



```

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

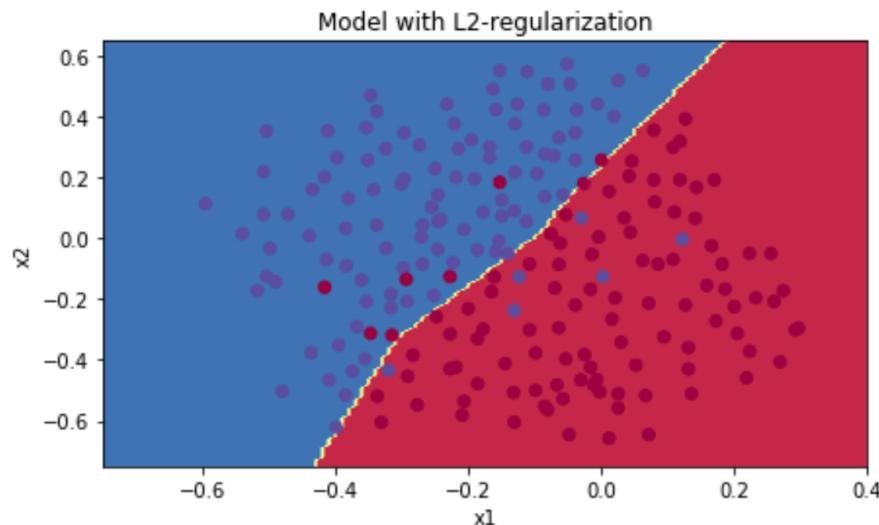
```

可以看出过拟合程度大大降低。我们再来看看决策边界

```

1 plt.title("Model with L2-regularization")
2 axes = plt.gca()
3 axes.set_xlim([-0.75, 0.40])
4 axes.set_ylim([-0.75, 0.65])
5 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
train_Y)

```



你也可以自己调整  $\lambda$  试试，过大的  $\lambda$  会使得曲线过于平滑，接近直线。这个方法认为，小的权重值比大的权重值使得模型更简单，因为它惩罚平方范数，使得权重变得更小（不是非常明白这里的意思，），总之，最终权重会变得很小。

预防过拟合的另一个方法是使用 dropout 方法，这个方法是在每次迭代中删除每个隐藏中的一些节点。通过观看文件夹中的两个视频，应该可以比较直观的理解这个算法。两个视频分别

为：[video/dropout1\\_kiank.mp4](#) 和 [video/dropout2\\_kiank.mp4](#) (visual studio code 自带的 markdown 预览器能够查看视频，但是会显示不出数学公式，markdown preview enhanced 这个扩展能够正常显示数学公式，但是加载不出来视频)。第二个视频中，第一层保留节点的概率是 0.6，第二层是 1.0，第三层是 0.8。

dropout 的想法是，在每次迭代中使用原先神经元中的一组神经元构成一个小的神经网络，这使得一组神经元对另一个神经元的激活不会那么敏感。

为三层神经网络部署 dropout 算法，输入层和输出层是不需要部署 dropout 算法的，因此，部署过程分为以下几个步骤：

1. 构建一个随机向量  $D$ ，它的维度和  $A$  一致
2. 设定一个  $keep\_prob$  值，决定保留节点的概率（这里教程中给出的是： $X < 0.5$ ，测试中  $keep\_prob = 0.7$ ，可以这样理解，小于 0.7 的概率为 70%，大于则为 30%，所以这里用的是小于）。
3.  $A = A * D$ ，去除节点
4.  $A$  整体除以  $keep\_prob$ ，保持期望值不变

代码如下所示：

```

1 def forward_propagation_with_dropout(X, parameters, keep_prob = 0.5):
2
3     np.random.seed(1)
4
5     # retrieve parameters

```

```

6     W1 = parameters[ "W1" ]
7     b1 = parameters[ "b1" ]
8     W2 = parameters[ "W2" ]
9     b2 = parameters[ "b2" ]
10    W3 = parameters[ "W3" ]
11    b3 = parameters[ "b3" ]
12
13    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
14    Z1 = np.dot(W1, X) + b1
15    A1 = relu(Z1)
16    ##### START CODE HERE ##### (approx. 4 lines)           # Steps 1-4 below
correspond to the Steps 1-4 described above.
17    D1 = np.random.rand(A1.shape[0], A1.shape[1])           # Step 1:
initialize matrix D1 = np.random.rand(..., ...)
18    D1 = D1 < keep_prob                                     # Step 2:
convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
19    A1 = A1 * D1                                           # Step 3: shut
down some neurons of A1
20    A1 = A1 / keep_prob                                     # Step 4:
scale the value of neurons that haven't been shut down
21    ##### END CODE HERE #####
22    Z2 = np.dot(W2, A1) + b2
23    A2 = relu(Z2)
24    ##### START CODE HERE ##### (approx. 4 lines)
25    D2 = np.random.rand(A2.shape[0], A2.shape[1])
# Step 1: initialize matrix D2 = np.random.rand(..., ...)
26    D2 = D2 < keep_prob                                     # Step 2:
convert entries of D2 to 0 or 1 (using keep_prob as the threshold)
27    A2 = A2 * D2                                           # Step 3: shut
down some neurons of A2
28    A2 = A2 / keep_prob                                     # Step 4:
scale the value of neurons that haven't been shut down
29    ##### END CODE HERE #####
30    Z3 = np.dot(W3, A2) + b3
31    A3 = sigmoid(Z3)
32
33    cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)
34
35    return A3, cache

```

对于反向传播来说，有下面两个步骤：

1. 在前向传播中使用的 masks  $D^l$  应该被保留，并作用在  $dA^l$  上。
2. 由于前向传播中，对  $A^l$  进行了  $keep\_prob$  缩放，且在微积分中，被缩放的  $A^l$ ，其导数也应该被缩放，所以我们仍然要对  $dA^l$  进行  $keep\_prob$  的缩放。

具体代码如下段所示：

```
1 def backward_propagation_with_dropout(X, Y, cache, keep_prob):
```

```

2     m = X.shape[1]
3     (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache
4
5     dZ3 = A3 - Y
6     dW3 = 1./m * np.dot(dZ3, A2.T)
7     db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
8     dA2 = np.dot(W3.T, dZ3)
9     ### START CODE HERE ### (≈ 2 lines of code)
10    dA2 = dA2 * D2           # Step 1: Apply mask D2 to shut down the
11    same neurons as during the forward propagation
12    dA2 = dA2 / keep_prob      # Step 2: Scale the value of
13    neurons that haven't been shut down
14    ### END CODE HERE ###
15    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
16    dW2 = 1./m * np.dot(dZ2, A1.T)
17    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
18
19    dA1 = np.dot(W2.T, dZ2)
20    ### START CODE HERE ### (≈ 2 lines of code)
21    dA1 = dA1 * D1           # Step 1: Apply mask D1 to shut down the
22    same neurons as during the forward propagation
23    dA1 = dA1 / keep_prob      # Step 2: Scale the value of
24    neurons that haven't been shut down
25    ### END CODE HERE ###
26    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
27    dW1 = 1./m * np.dot(dZ1, X.T)
28    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
29
30    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
31                  "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
32                  "dZ1": dZ1, "dW1": dW1, "db1": db1}
33
34    return gradients

```

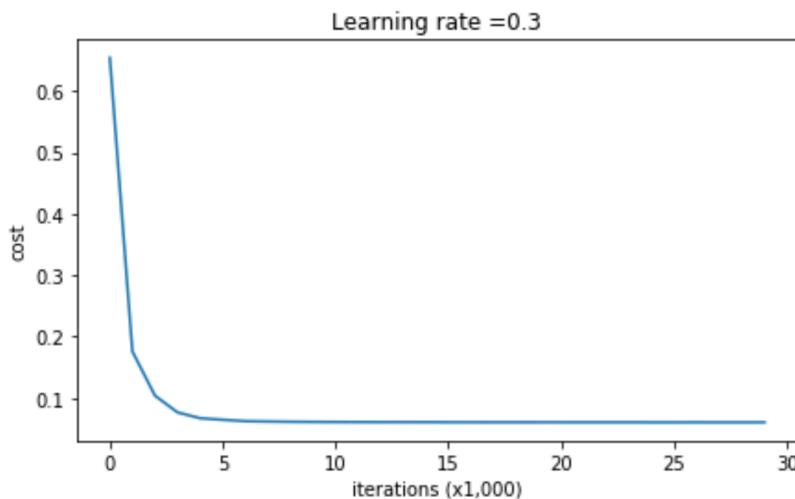
接下来，我们来测试这个模型的效果

```

1 parameters = model(train_X, train_Y, keep_prob = 0.86, learning_rate = 0.3)
2
3 print ("On the train set:")
4 predictions_train = predict(train_X, train_Y, parameters)
5 print ("On the test set:")
6 predictions_test = predict(test_X, test_Y, parameters)

```

```
Cost after iteration 0: 0.6543719293813615
Cost after iteration 10000: 0.06100575107627548
Cost after iteration 20000: 0.06057120083858194
```

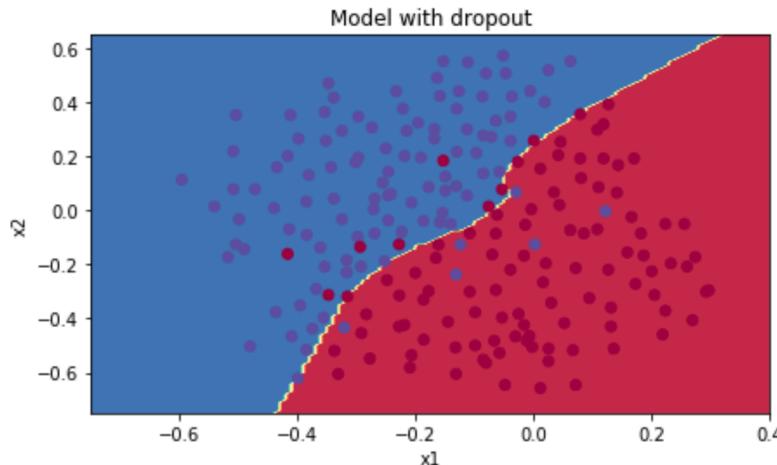


```
On the train set:  
Accuracy: 0.9289099526066351  
On the test set:  
Accuracy: 0.95
```

注意，这里我修改了教程 reg\_utils.py 文件中的 compute\_cost 函数，改动了 logprob 的计算方式：

```
1 | logprobs = np.multiply(-np.log(a3 + 1e-5),Y) + np.multiply(-np.log(1 - a3 +  
1e-5), 1 - Y)
```

这里如果不修改，计算 log 时会溢出。测试集拥有 95% 的准确率，看起来确实很棒。决策边界如下图所示：



下面是使用 dropout 的几个要点

1. dropout 只用于训练集，千万别在测试集用它
2. 使用 dropout 时，别忘了做 keep\_prob 判断
3. 使用 dropout 时，要对前向传播和反向传播都做一次 dropout
4. dropout 是一个正则化方法！

归一化处理

关于神经网络训练的加速，还有一种可行的办法，就是对输入的数据集做归一化处理，至于原因，会在讲解完归一化处理方法后给出。

教程中的归一化实际上是概率论中标准化处理的效果。这里说是效果，是因为作者给出的计算方法，和作者给出的处理结果不一致。对于输入训练集  $X$ ，作者的处理方式如下：

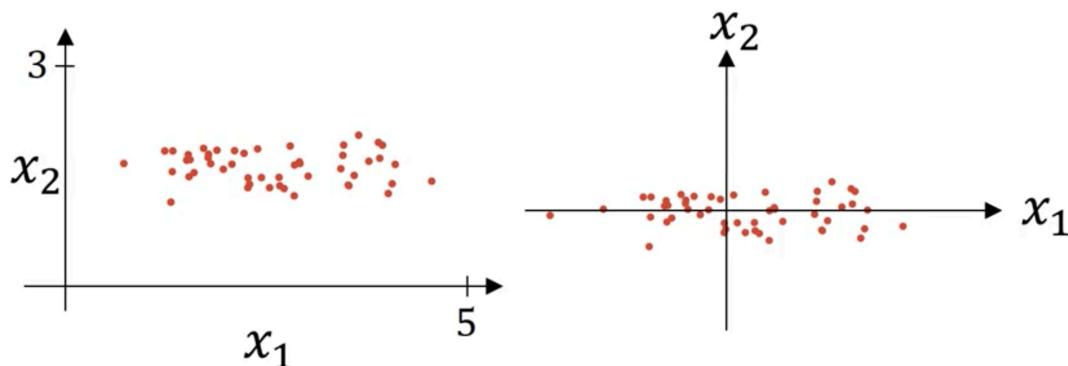
$$\mu = \frac{1}{m} \sum_{i=1}^m X^{(i)} \quad (1)$$

$$X' = X - \mu \quad (2)$$

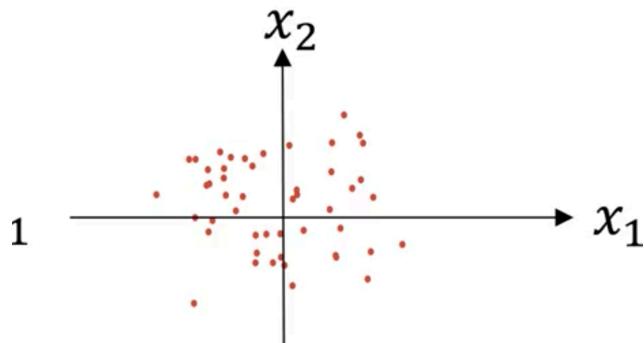
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m X'^{(i)2} \text{ (此处为平方)} \quad (3)$$

$$X = \frac{X'}{\sigma^2} \quad (4)$$

公式 2 的处理效果为，将训练集数据中心偏移到坐标轴原点，图示如下：



公式 4 的处理效果为：使  $x_1$  和  $x_2$  的方差为 1，图示如下：



但是我根据作者的步骤计算，并不能够得出方差为 1 的结果，代码如下：

```
1 import numpy as np
2
3 def variance(a):
4     mean = np.mean(a)
5     print("mean = " + str(mean))
6     b = a - mean
7     c = np.square(b)
8     c = np.mean(c)
9     print("variance = " + str(c));
10    # c = np.sqrt(c) # 采用作者的方法，注释此行
```

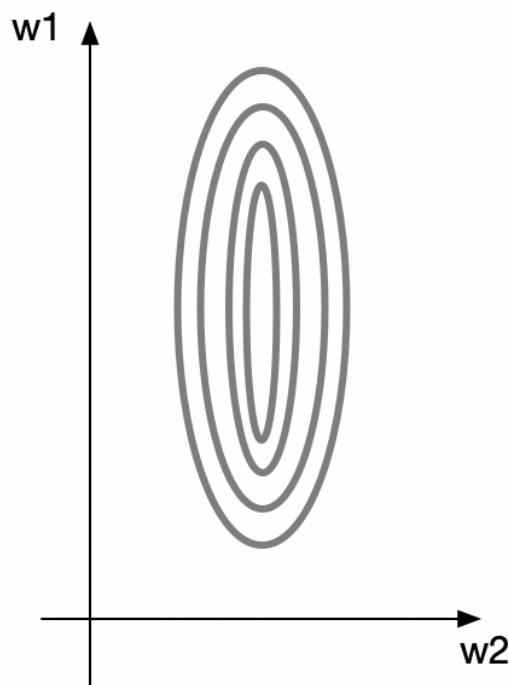
```

11     return (b/c)
12
13 a = np.random.randn(1, 5)
14 b = variance(a)
15 variance(b)

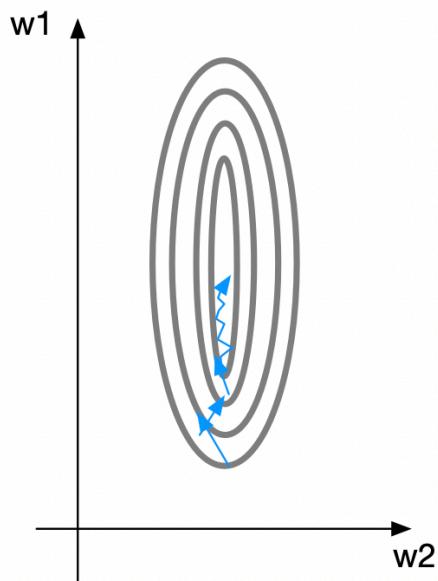
```

若将 `# c = np.sqrt(c)` 去除注释，则可以得到方差为 1.0 或 0.999999... 的结果。所以我认为这里实际上是在计算标准差。

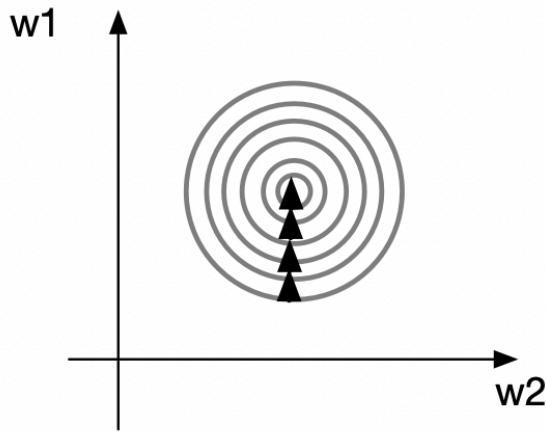
那么，上述方法为什么能够缩短训练时间呢？考虑一个训练数据，它的特征向量有两个特征值  $x_1 \in [1, 1000]$  和  $x_2 \in [0, 1]$ 。那么在训练后得到的参数  $w_1$  和  $w_2$  之间的范围差距会很大，损失函数  $J$  关于  $w_1$  和  $w_2$  在  $w_1$ ,  $w_2$  轴上的投影图像就如下图所示：



这会使得学习率必须选择一个较小的值，并且会使得从最外层椭圆上任意一点出发的梯度下降变得缓慢，如下图所示：



若进行归一化，则  $w_1$  和  $w_2$  成比例，图像变成这样：

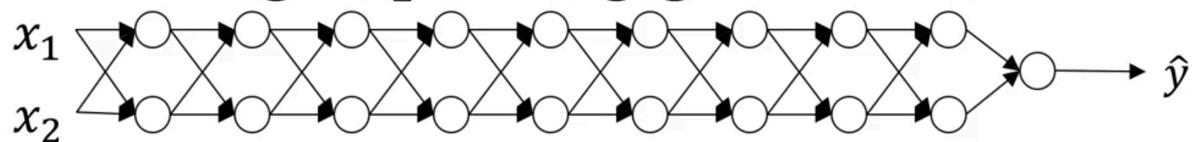


不仅能够提高学习率，而且从任意一点都能很快收敛到最小值处。

对于本身数据就在相似比例附近的训练集，进行归一化并不会提升训练速度，但是归一化处理本身并没有什么坏处。

### 梯度爆炸/消失 (vanishing/exploding of gradients) 问题

下图为一个特征向量只有两个数据值的深度神经网络



我们假设激活函数是线性的  $g^{[l]}(z^{[l]}) = z^{[l]}$ ，每个参数  $w^{[l]}$  均为： $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ ，参数  $b^{[l]} = 0$ ， $\hat{y}$  的计算如下

$$\hat{y} = w^{[L]} w^{[l-1]} w^{[l-2]} \dots w^{[2]} w^{[1]} x$$

这里就不推导公式了。根据对参数  $w^{[l]}$  的赋值，我们可以最后得出：

$$\hat{y} = w^{[l]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{[l-1]} x$$

因为  $x$  是二维矩阵，所以最后  $\hat{y} = 1.5^L x$ （教程中这里写的是  $1.5^{[L-1]}x$ ，我特地去找了比较靠谱的 [summary](#)，笔记里写的就是  $\hat{y} = 1.5^L x$ ）。

假如  $L$  非常大，那么可以看出  $1.5^L$  会变得非常大，这被称为梯度爆炸，不仅会导致计算机无法处理这类超大数据，还会导致无法训练模型。假如  $w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$  时，若  $L$  非常大，则会导致  $0.5^L$  非常小，这被称作梯度消失，同样会导致计算机无法处理这类超小数据，以及训练模型速度很慢。下一节内容会详细说明如何解决这个问题。

### 针对梯度爆炸/梯度消失进行的权重初始化

之前我们提到了梯度爆炸和梯度消失的坏处，那么选择合适的权重值是解决该问题的最好方法，它会使得梯度爆炸和梯度消失的速度变得缓慢（但不能够完全解决这个问题）。之前提到过，为了避免对称性问题，初始化可以使用高斯函数随机变量，但这仍然会导致数据溢出，这在上一个编程练习中是提到过的。如果你还记得 He initialization，初始化时对随机获得的权重后面乘上的 `np.sqrt(2/n[1-1])` 就是被验证的，用于  $g(z) = ReLU(z)$  的，能有效降低梯度爆炸/梯度下降的方法（定语真多。。。）。作者在视频中的解释为：`np.sqrt(2/n[1-1])` 将原先的方差变为  $2/n$ （ $n$  为  $n^{[l-1]}$ ）

层特征向量的特征个数），这样能够使得在激活函数为 ReLU 时，权重值较为合适。同样有论文表明，在使用激活函数为  $\tanh(z)$  时，方差为  $1/n$  比较合适，另一篇论文则表明，对于  $\tanh$ ，方差取  $1/(n^{[l-1]} + n^{[l]})$  也可行。

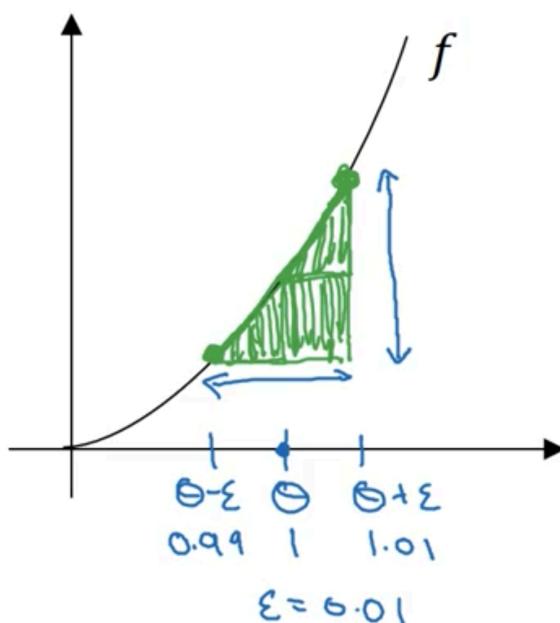
同样，你也可以为 `np.sqrt(2/n[1-1])` 添加一个乘量，作为超参数，但相比于其他超参数，该参数优先级还是比较低的。

### 梯度检测的数值逼近（针对梯度的数值逼近）

在进行反向传播的时候，尽管写下了公式，但是不能够 100% 确定程序在进行完全正确的反向传播，因此需要做梯度检测。实现梯度检测需要用到针对梯度的数值逼近。

从这篇[博文](#)来看，我认为数值上导数和梯度是相同的（实际上梯度值是方向导数最大值，该值对应的方向为梯度方向）。这里就以导数为例，研究数值逼近。[（这里用导数研究梯度的原因视频中没有提出，如果我的观点有误，请提出）](#)

视频中是先给出了一个三次函数  $f(\theta) = \theta^3$ ，然后从双边公差（two side difference）入手进行研究，如下图所示：



我们按照导数的定义去计算高宽比值的话，可以得到  $(f(0.99) - f(1.01))/0.02 = 3.0001$ ，与  $f'(1) = 3$  只相差 0.0001。如果我们使用单边计算，可以得到  $f((1.01) - f(1))/0.01 = 3.0301$ 。这个误差比双边误差要高很多。我们也可以用导数定义来理解这个误差，这里可以参考这个[知乎答案](#)的推导来理解视频中对单边误差为  $O(\epsilon)$ ，双边误差为  $O(\epsilon^2)$  的结论。下一节我们讲讲如何将这个方法应用在梯度检测中。

### 应用梯度检测

上一节讲了梯度检测的方法，这里我们来应用梯度检测。

对于初始化的参数  $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$ ，将所有的  $W$  矩阵 reshape 成向量，然后将所有的参数（所有层的  $W$  和  $b$ ）组合起来，变成一个巨大的向量  $\theta$ ，那么成本函数  $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]})$  就变成了  $J(\theta)$ 。这里  $\theta$  的实现思路在编程练习 3 中会明确说明。

我们按照同样的方法处理参数导数  $dW^{[1]}, db^{[1]}, dW^{[2]}, db^{[2]}, \dots, dW^{[L]}, db^{[L]}$ ，将他们变成一个巨大的向量  $d\theta$ 。我们接下来要做的，就是计算双向误差以及导数和双向误差之间的距离。

对于  $J(\theta)$ ，可以展开为  $J(\theta_1, \theta_2, \dots, \theta_L)$ 。我们设置一个向量  $d\theta_{approx}$  记录双向误差，对该向量进行如下赋值：

For each  $i$  :

$$d\theta[i]_{approx} = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

上述伪代码意在计算双边误差，或者说是  $\theta$  偏导数的近似值，接下来，我们用欧几里得范数来衡量反向传播计算的参数导数和双边误差之间的差距，公式如下：

$$dis = \frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

向量欧式距离就是向量中的各个数据的平方和，再开根。这里的分母  $\|d\theta_{approx}\|_2 + \|d\theta\|_2$  是为了防止分子太小或太大而设置的。验证的方法为：

当  $\epsilon = 10^{-7}$  时

1. 若  $dis$  在  $10^{-7}$  附近，则说明反向传播是成功的，
2. 若在  $10^{-5}$  附近，则说明可能是错误的，可能是正确的，
3. 若大于  $10^{-5}$ ，那么很可能你的反向传播是错误的。

### 编程练习前需要注意的训练技巧

1. 梯度检测只用于验证，不要将它用于训练（梯度检测时间成本不低）
2. 当你使用 dropout 时，不要使用梯度验证（dropout 会使代价函数不容易计算，如果要用，建议是先梯度检测，关闭 dropout，梯度检测结束再打开 dropout）
3. 现实中很少但可能会出现当参数被初始化为很接近 0 的值时，反向传播算法是正确的，但在慢慢增大的后，反向传播的算法就不正确了。不常用的解决办法是在随机初始化时，运行梯度检测，这使得  $W$  和  $b$  会有一段时间远离 0，反复训练之后在进行梯度检测（这里不太理解）。
4. 如果在梯度检测时找到了 bug，在每一层中仔细查找  $dw$  和  $db$  的值，这里不理解的话没关系，编程练习 3 中会讲到。
5. 记得如果你实施了正则化，一定要在求导的时候也考虑正则化！

### 课后编程练习 3

本周课程的最后一个编程练习内容就是实现梯度检测，作者设置的背景也很有趣，大意为：你是检测手机全球支付欺诈团队成员之一，由于该任务很重要，因此你需要对你们团队的神经网络模型进行梯度检测，确保模型正确实施，让老板放心。

执行梯度检测的环境如下

```
1 # Packages
2 import numpy as np
3 from testCases import *
4 from gc_utils import sigmoid, relu, dictionary_to_vector,
vector_to_dictionary, gradients_to_vector
```

`testCases.py` 和 `gc_utils.py` 文件均被放置在 `Week1_编程练习/编程练习3` 中的 `functions` 文件夹下。

对于 1 维的梯度检测，假设  $J(\theta) = \theta x$ ,  $x$  为训练集，那么前向传播的代码如下：

```

1 def forward_propagation(x, theta):
2     """
3         Implement the linear forward propagation (compute J) presented in
4         Figure 1 ( $J(\theta) = \theta * x$ )
5         Arguments:
6             x -- a real-valued input
7             theta -- our parameter, a real number as well
8         Returns:
9             J -- the value of function J, computed using the formula  $J(\theta) =$ 
10             $\theta * x$ 
11            """
12
13    #### START CODE HERE #### (approx. 1 line)
14    J = theta * x
15
16    #### END CODE HERE ####
17
18    return J

```

接下来，进行反向传播，对  $\theta$  求偏导数，代码如下：

```

1 def backward_propagation(x, theta):
2     """
3         Computes the derivative of J with respect to theta (see Figure 1).
4         Arguments:
5             x -- a real-valued input
6             theta -- our parameter, a real number as well
7         Returns:
8             dtheta -- the gradient of the cost with respect to theta
9             """
10
11    #### START CODE HERE #### (approx. 1 line)
12    dtheta = forward_propagation(x, theta) / theta
13
14    #### END CODE HERE ####
15
16    return dtheta

```

为了验证反向传播是否正确，我们需要按照下述步骤实现梯度检测：

1.  $\theta^+ = \theta + \epsilon$
2.  $\theta^- = \theta - \epsilon$
3.  $J^+ = J(\theta^+)$
4.  $J^- = J(\theta^-)$
5.  $gradapprox = \frac{J^+ - J^-}{2\epsilon}$
6. 计算反向传播的梯度
7.  $difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2}$

代码如下：

```

1 # GRADED FUNCTION: gradient_check
2
3 def gradient_check(x, theta, epsilon = 1e-7):
4     """

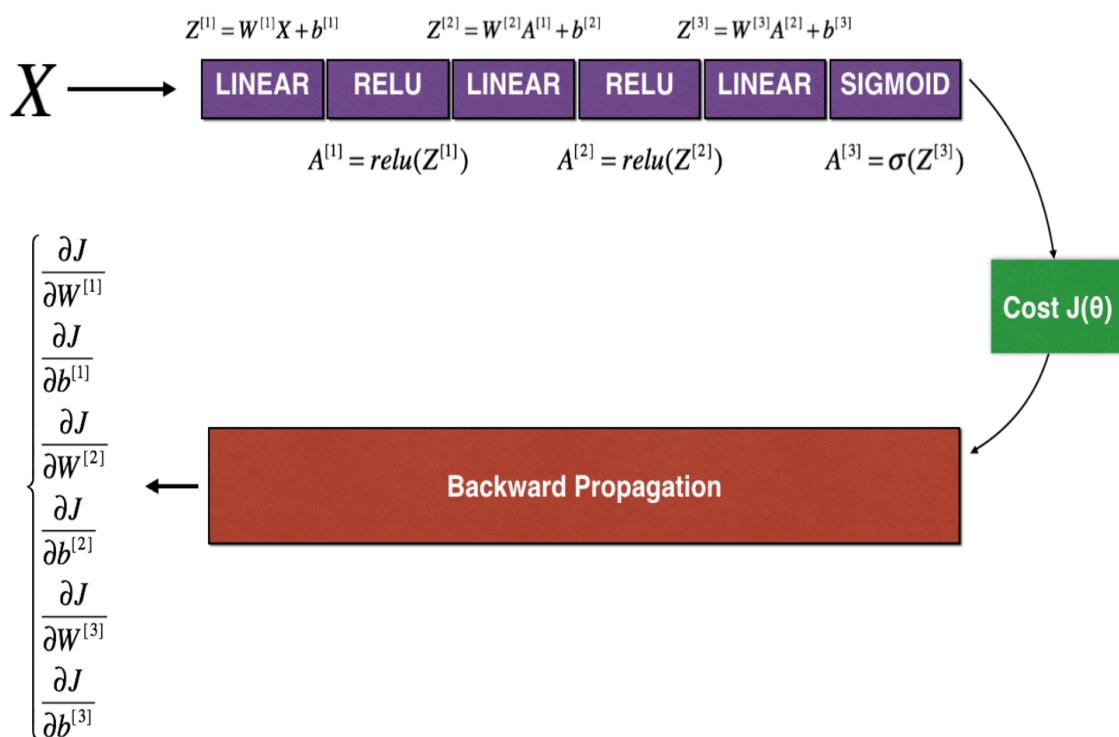
```

```

5     Implement the backward propagation presented in Figure 1.
6
7     Arguments:
8         x -- a real-valued input
9         theta -- our parameter, a real number as well
10        epsilon -- tiny shift to the input to compute approximated gradient
11        with formula(1)
12
13    Returns:
14        difference -- difference (2) between the approximated gradient and the
15        backward propagation gradient
16
17        """
18
19
20        # Compute gradapprox using left side of formula (1). epsilon is small
21        enough, you don't need to worry about the limit.
22
23        ### START CODE HERE ### (approx. 5 lines)
24        thetaplus = theta + epsilon                      # Step 1
25        thetaminus = theta - epsilon                     # Step 2
26        J_plus = forward_propagation(x, thetaplus)      # Step 3
27        J_minus = forward_propagation(x, thetaminus)    # Step 4
28        gradapprox = (J_plus - J_minus) / (2*epsilon)    # Step 5
29
30        ### END CODE HERE ###
31
32        # Check if gradapprox is close enough to the output of
33        backward_propagation()
34
35        ### START CODE HERE ### (approx. 1 line)
36        grad = backward_propagation(x, theta)
37
38        ### END CODE HERE ###
39
40        ### START CODE HERE ### (approx. 1 line)
41        numerator = np.linalg.norm(grad - gradapprox)
42
43        # Step 1'
44        denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
45
46        # Step 2'
47        difference = numerator/denominator
48
49        # Step 3'
50
51        ### END CODE HERE ###
52
53        if difference < 1e-7:
54            print ("The gradient is correct!")
55        else:
56            print ("The gradient is wrong!")
57
58        return difference

```

对于 N 维的梯度检测，过程如下图所示：



**Figure 2**: deep neural network

\*LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID\*

图中步骤很明确：

1. 计算各个参数和中间结果
2. 计算成本函数
3. 反向传播计算导数
4. 梯度检测

先来计算正向传播，和反向传播，作者给出的代码如下：

```

1 def forward_propagation_n(X, Y, parameters):
2     # retrieve parameters
3     m = X.shape[1]
4     W1 = parameters["W1"]
5     b1 = parameters["b1"]
6     W2 = parameters["W2"]
7     b2 = parameters["b2"]
8     W3 = parameters["W3"]
9     b3 = parameters["b3"]
10
11    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
12    Z1 = np.dot(W1, X) + b1
13    A1 = relu(Z1)
14    Z2 = np.dot(W2, A1) + b2
15    A2 = relu(Z2)
16    Z3 = np.dot(W3, A2) + b3
17    A3 = sigmoid(Z3)
18    # Cost

```

```

19     logprobs = np.multiply(-np.log(A3), Y) + np.multiply(-np.log(1 - A3), 1
- Y)
20     cost = 1./m * np.sum(logprobs)
21     cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)
22     return cost, cache

```

```

1 def backward_propagation_n(X, Y, cache):
2     m = X.shape[1]
3     (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache
4
5     dZ3 = A3 - Y
6     dW3 = 1./m * np.dot(dZ3, A2.T)
7     db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
8
9     dA2 = np.dot(W3.T, dZ3)
10    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
11    dW2 = 1./m * np.dot(dZ2, A1.T) * 2
12    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
13
14    dA1 = np.dot(W2.T, dZ2)
15    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
16    dW1 = 1./m * np.dot(dZ1, X.T)
17    db1 = 4./m * np.sum(dZ1, axis=1, keepdims = True)
18
19    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
20                  "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
21                  "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}
22
23    return gradients

```

下一步就是将获得的参数转换成一个大向量，这里作者替我们写好了函数，针对代码来理解一下思路：

```

1 def dictionary_to_vector(parameters):
2     # 由于练习中使用 parameters 存储所有变量，因此我们要做的就是将 parameters 变成
3     # 一个大向量
4
5     # keys 用于存储大向量的字典序列，更新逻辑可以在循环中体现出来。
6     keys = []
7     # count 考虑原先参数的个数，逻辑在循环中体现
8     count = 0
9     for key in ["W1", "b1", "W2", "b2", "W3", "b3"]:
10         # 将 parameters 中的所有参数均变成 (x,1) 维度向量，x 由 python 根据参数自
11         # 动推测，存储在 new_vector 中
12         new_vector = np.reshape(parameters[key], (-1, 1))
13         # [key] * new_vector.shape[0] 将返回 new_vector.shape[0] 个 [key]
14         # 例如 [key] 为 'W1'，new_vector.shape[0] 为 3，则：
15         # [key] * new_vector.shape[0] 为 ['W1', 'W1', 'W1']
16         # keys 依次记录所有这样的 key

```

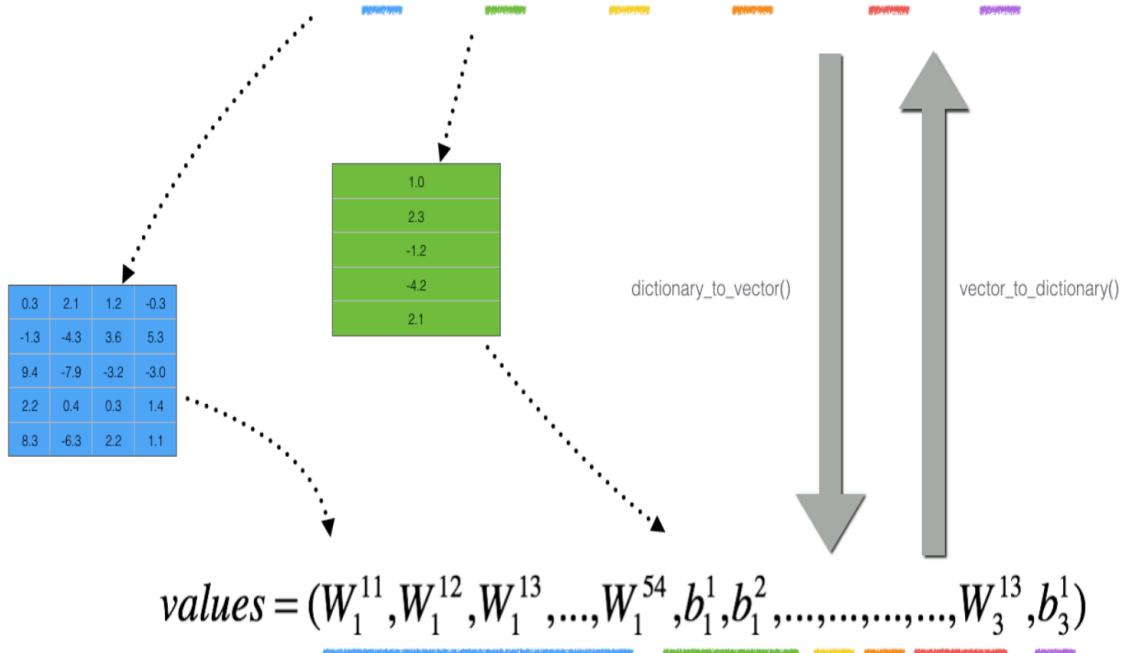
```

15     keys = keys + [key] * new_vector.shape[0]
16
17     # 只有一个参数时, theta 被初始化为 new_vector
18     if count == 0:
19         theta = new_vector
20     else: # 有多个参数时, 用 np.concatenate() 把它们拼接成一列
21         theta = np.concatenate((theta, new_vector), axis=0)
22     count = count + 1
23
24
25     return theta, keys

```

结合下面的图片，应该能更好理解上面的代码：

$parameters = \{ "W_1": ..., "b_1": ..., "W_2": ..., "b_2": ..., "W_3": ..., "b_3": ... \}$



\*\*Figure 2\*\* : \*\*`dictionary_to_vector()` and `vector_to_dictionary()`\*\*

You will need these functions in `gradient_check_n()`

接下来我们就根据步骤，实现 n 维的梯度检测函数：

对于每一个参数：

- 计算 `J_plus[i]`：
  - 将  $\theta^+$  设置为 `np.copy(parameters_values)`
  - 将  $\theta_i^+$  设置为  $\theta_i^+ + \varepsilon$
  - 用 `forward_propagation_n(x, y, vector_to_dictionary(theta^+))` 计算  $J_i^+$ ，
- 按照类似的方法计算 `J_minus[i]`
- 计算  $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

```

1 # GRADED FUNCTION: gradient_check_n
2
3 def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7):
4     """

```

```

5     Checks if backward_propagation_n computes correctly the gradient of
6     the cost output by forward_propagation_n
7
8     Arguments:
9         parameters -- python dictionary containing your parameters "W1", "b1",
10        "W2", "b2", "W3", "b3":
11        grad -- output of backward_propagation_n, contains gradients of the
12        cost with respect to the parameters.
13        x -- input datapoint, of shape (input size, 1)
14        y -- true "label"
15        epsilon -- tiny shift to the input to compute approximated gradient
16        with formula(1)
17
18        Returns:
19        difference -- difference (2) between the approximated gradient and the
20        backward propagation gradient
21        """
22
23
24    # Set-up variables
25    parameters_values, _ = dictionary_to_vector(parameters)
26    grad = gradients_to_vector(gradients)
27    num_parameters = parameters_values.shape[0]
28    J_plus = np.zeros((num_parameters, 1))
29    J_minus = np.zeros((num_parameters, 1))
30    gradapprox = np.zeros((num_parameters, 1))
31
32    # Compute gradapprox
33    for i in range(num_parameters):
34
35        # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output
36        = "J_plus[i]".
37        # "_" is used because the function you have to outputs two
38        parameters but we only care about the first one
39        ### START CODE HERE ### (approx. 3 lines)
40        thetaplus = np.copy(parameters_values)
41        # Step 1
42        thetaplus[i][0] = thetaplus[i][0] + epsilon
43        # Step 2
44        J_plus[i], _ = forward_propagation_n(X, Y,
45        vector_to_dictionary(thetaplus))                                # Step
46        3
47
48        ### END CODE HERE ###
49
50
51        # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output
52        = "J_minus[i]".
53        ### START CODE HERE ### (approx. 3 lines)
54        thetaminus = np.copy(parameters_values)
55        # Step 1

```

```

40         thetaminus[i][0] = thetaminus[i][0] - epsilon
        # Step 2
41         J_minus[i], _ = forward_propagation_n(X, Y,
vector_to_dictionary(thetaminus))                                     # Step
42
43             ### END CODE HERE ###
44
45             # Compute gradapprox[i]
46             ### START CODE HERE ### (approx. 1 line)
47             gradapprox[i] = (J_plus[i] - J_minus[i]) / (2*epsilon)
48             ### END CODE HERE ###
49
50             # Compare gradapprox to backward propagation gradients by computing
difference.
51             ### START CODE HERE ### (approx. 1 line)
52             numerator = np.linalg.norm(grad - gradapprox)
                # Step 1'
53             denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
                # Step 2'
54             difference = numerator / denominator
                # Step 3'
55             ### END CODE HERE ###
56
57             if difference > 2e-7:
58                 print ("\u033[93m" + "There is a mistake in the backward
propagation! difference = " + str(difference) + "\u033[0m")
59             else:
60                 print ("\u033[92m" + "Your backward propagation works perfectly
fine! difference = " + str(difference) + "\u033[0m")
61
62             return difference

```

运行教程中给出的测试代码：

```

1 X, Y, parameters = gradient_check_n_test_case()
2 cost, cache = forward_propagation_n(X, Y, parameters)
3 gradients = backward_propagation_n(X, Y, cache)
4 difference = gradient_check_n(parameters, gradients, X, Y)

```

会输出： There is a mistake in the backward propagation! difference =

0.2850931567761624

输出信息告诉我们出错了，因此我们需要检查 backward\_propagation\_n 函数中有没有错误，经过检查发现下列参数的计算出现错误：

```

1 dZ3 = A3 - Y
2 dW3 = 1./m * np.dot(dZ3, A2.T)
3 db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
4

```

```

5     dA2 = np.dot(W3.T, dZ3)
6     dZ2 = np.multiply(dA2, np.int64(A2 > 0))
7 #     dW2 = 1./m * np.dot(dZ2, A1.T) * 2
8     dW2 = 1./m * np.dot(dZ2, A1.T)
9     db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
10
11    dA1 = np.dot(W2.T, dZ2)
12    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
13    dW1 = 1./m * np.dot(dZ1, X.T)
14 #     db1 = 4./m * np.sum(dZ1, axis=1, keepdims = True)
15     db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

```

被注释掉的行为错误行，改正后再次运行则提示正确。

## Week 2 优化算法

### mini-batch 算法

之前加速训练的手段，例如正则化输入值或是 dropout，其实还不够。试想一下，当你的数据集在 5,000,000 量级，而一次梯度下降仅仅能够计算一次参数更新，这样的效率是比较低的。为了解决这个问题，mini-batch 算法将训练集划分成等距离的若干个小训练集，对其进行梯度下降（也叫 batch 梯度下降，对应下面要讲的 mini-batch），这样能够成倍增多梯度下降次数。我们来看看这个算法。

假设你有 5,000,000 数量级的训练集，包括数据  $X$  和标签  $Y$ ，如下所示：

$$X(n_x, 5,000,000) = \{x^{(1)}, x^{(2)}, \dots, x^{(5,000,000)}\}$$

$$Y(1, 5,000,000) = \{y^{(1)}, y^{(2)}, \dots, y^{(5,000,000)}\}$$

接着我们每隔 1000 个训练数据划分一次，如下所示：

$$X(n_x, 5,000,000) = \underbrace{\{x^{(1)}, x^{(2)}, \dots, x^{(1000)}\}}_{X^{\{1\}}}, \underbrace{\{x^{(1001)}, \dots, x^{(2000)}\}}_{X^{\{2\}}}, \dots, \underbrace{\{x^{(5,000,000)}\}}_{X^{\{5000\}}}$$

$$Y(n_x, 1) = \underbrace{\{y^{(1)}, y^{(2)}, \dots, y^{(1000)}\}}_{Y^{\{1\}}}, \underbrace{\{y^{(1001)}, \dots, y^{(2000)}\}}_{Y^{\{2\}}}, \dots, \underbrace{\{y^{(5,000,000)}\}}_{Y^{\{5000\}}}$$

我们用  $\{t\}$  表示 mini-batch 的第几个数据，在训练数据的时候，伪代码如下所示：

for  $t$  in 5000 :

forward prop on  $X^{[t]}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

...

$$Z^{[L]} = W^{[L]} X^{\{t\}} + b^{[L]}$$

$$A^{[L]} = g^{[L]}(Z^{[L]})$$

$$J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} L(\hat{y}^{\{t\}(i)}, y^{\{t\}(i)}) + \underbrace{\frac{\lambda}{2 * 1000} + \sum_l \|w^l\|_F}_{optional}$$

backward prop on  $J^{\{t\}}$  using  $(w^{\{t\}}, b^{\{t\}})$

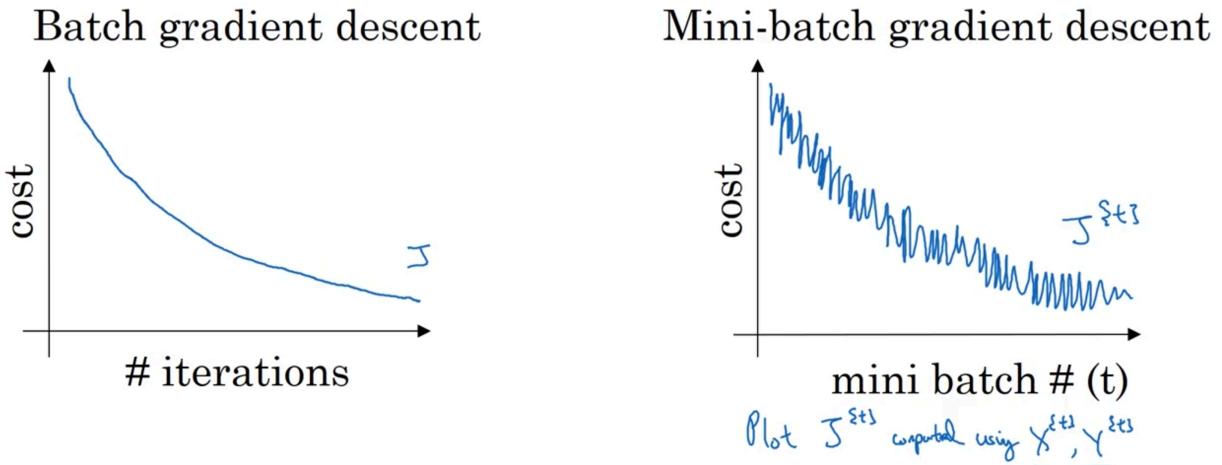
$$w^{[l]} := w^{[l]} - \alpha d w^{[l]}$$

$$b^{[l]} := b^{[l]} - \alpha d b^{[l]}$$

可以看出，参数在一次完整的训练集中直接被训练了 5000 次，forward prop on  $X^{\{t\}}$  的过程可以用向量化完成。我们会在下节内容详细讲解该方法。

### 理解 mini-batch

对于梯度下降中的  $J$  值图像，batch 和 mini-batch 是不同的，如下图所示：



Andrew Ng

batch 梯度下降的图像不用解释了，那么 mini-batch 梯度下降的图像如上的原因是，由于训练集被分成了若个个小训练集，每个训练集计算出的成本函数的值都不太一致（教程中表示各个训练集计算成本函数的难易程度不一样，导致成本不一样）。

在之前的例子中，我们将 5,000,000 的数据集划分成了 5000 份，每份 1000 个训练数据。一份训练数据究竟取到多少比较合适，是一个值得探讨的问题。我们从两个极端来看。

我们将一份训练数据的个数用 size 表达，两个极端及其优劣为：

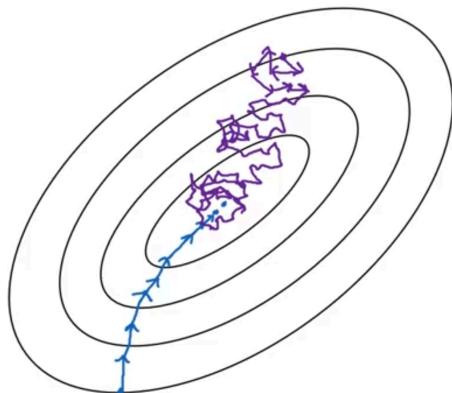
1. size = m

这表示我们取整个训练集为一个 mini-batch，实际上就是 batch 梯度下降。在数据量较少时，使用 batch 梯度下降能够快速有效的找到成本函数的最小值，而数据量较大时，该方法效率很低。

2. size = 1

这表示我们取一个数据作为一个 mini-batch，这个被称作随机梯度下降 (Stochastic gradient decent) 算法。由于数据集中只有一个数据，因此该算法会使得数据的噪声很严重，在梯度下降过程中，偏离梯度下降方向的几率恒大，最终也只会取到最小值周围的值，无法取到最小值。

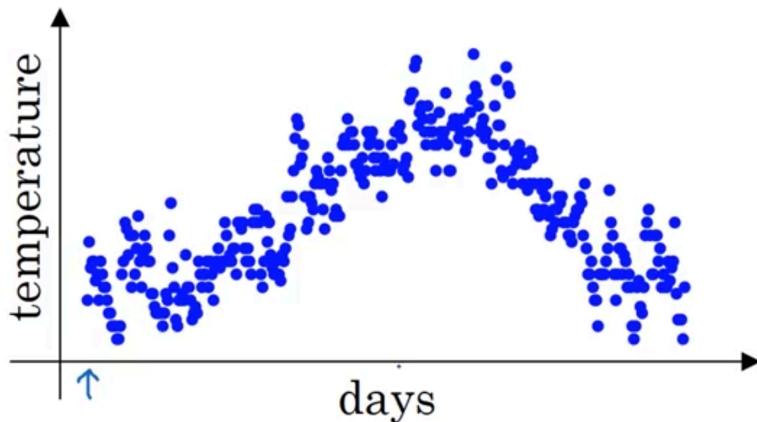
两种极端在梯度下降的表现图如下：



紫色表示  $\text{size} = 1$  的 mini-batch, 蓝色表示  $\text{size} = m$  的 mini-batch。我们很直观的可以发现, 只要取得  $1 < \text{size} < m$  即可。实验表示, size 取 2 的  $n$  次方效率比较高, 一般的范围在 64~512 之间, 视具体情况而定。当 mini-batch 的噪声太多时, 可以尝试降低学习率, 来减少噪声。当然, 这个尺寸还与 CPU/GPU 的内存相关, 如果你的 size 不符合 CPU/GPU 的内存, 那么算法效率令人堪忧。下一节会讲解比 batch 和 mini-batch 更高效的算法。

### 指数加权平均 (exponentially weighted averages, EWA)

接下来要学习的几个优化算法的基础是加权指数平均。因此我们需要先学习该算法。教程中, 作者用地区温度举例, 如下图:



由于数据比较杂乱, 如果要计算趋势的话, 可以计算加权平均值。作者在这里使用了指数加权平均的算法, 具体算法如下:

假定有 100 天的温度数据  $\theta_1, \theta_2, \dots, \theta_{100}$ , 权值  $\beta = 0.9$ , 有:

$$\begin{aligned} v_0 &= 0 \\ v_1 &= \beta * v_0 + (1 - \beta) * \theta_1 \\ &\dots \\ v_{100} &= \beta * v_{99} + (1 - \beta) * \theta_{100} \end{aligned}$$

这里有几个定义, 首先  $\beta = 0.9$ , 表示  $v_t$  ( $t$  表示天数) 为近  $1/(1 - 0.9)$  天的平均温度, 这个定义是根据  $0.9^n \approx 1/e$ ,  $n = 10$  来确定的。总结一下就是  $v_t$  表示  $1/(1 - \beta)$  天的平均温度。

那么明明叫指数加权, 指数体现在哪儿呢? 我们对四天的数据进行一下推算, 过程如下:  
为了方便起见, 我们取 4 天数据进行计算

$$\begin{aligned}
 v_0 &= 0 \\
 v_1 &= \beta * v_0 + (1 - \beta) * \theta_1 \\
 v_2 &= \beta * v_1 + (1 - \beta) * \theta_2 \\
 v_3 &= \beta * v_2 + (1 - \beta) * \theta_3 \\
 v_4 &= \beta * v_3 + (1 - \beta) * \theta_4
 \end{aligned}$$

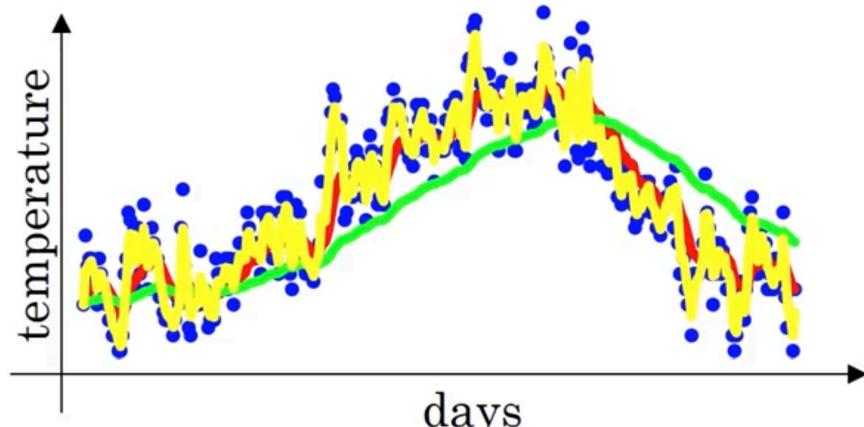
则  $v_4$  可以写成：

$$v_4 = \beta^4 v_0 + \beta^3 (1 - \beta) \theta_1 + \beta^2 (1 - \beta) \theta_2 + \beta (1 - \beta) \theta_3 + (1 - \beta) \theta_4$$

可以发现，其通式为：

$$\begin{aligned}
 v_t &= \beta^t v_0 + \beta^{(t-1)} (1 - \beta) \theta_1 + \dots + \beta (1 - \beta) \theta_{t-1} + (1 - \beta) \theta_t \\
 v_t &= (1 - \beta) \sum_{i=0}^{t-1} \beta^i \theta_{t-i}
 \end{aligned}$$

教程中分别对  $\beta = 0.9$ ,  $\beta = 0.98$ ,  $\beta = 0.5$  绘制了图像，颜色分别为红色，绿色和黄色



绿色整体向右偏移是因为  $v_t = (1 - \beta) \sum_{i=0}^{t-1} \beta^i \theta_{t-i}$  中，天数越靠后，权重值越大 ( $\beta < 1.0$ )。可以发现， $\beta$  越小，曲线越接近数据集原来的走势图，但是较为曲折，相反，则较为平坦，但难以拟合原先的数据。因此对于  $\beta$  的选取，应当介于两者之间。

我自己也做了尝试，代码如下：

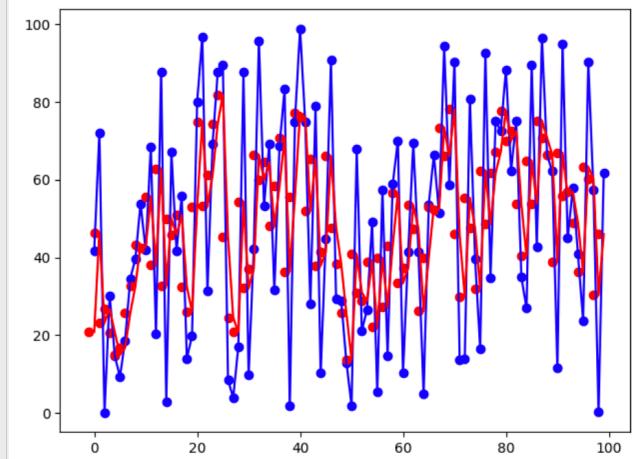
```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 dimension = 100
5 np.random.seed(1)
6 a = np.random.rand(1, dimension) * 100
7
8
9 vt = 0
10 vt_list = []
11 beta = 0.5
12 for i in range(dimension):
13     vt = vt * beta + (1. - beta) * a[0][i]
14     vt_list.append(vt)
15
16 b = np.array(range(-1, dimension-1))
17 c = np.array(vt_list)

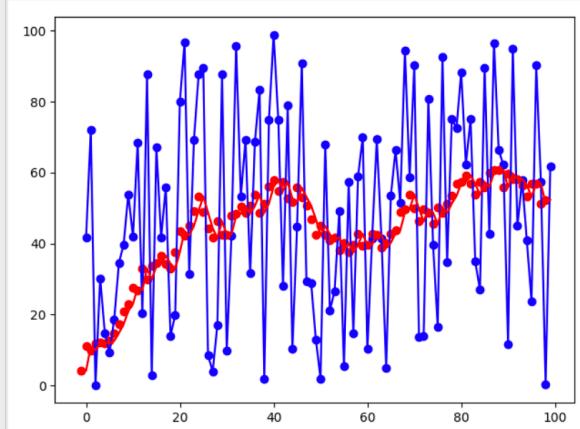
```

```
18 print(b.shape)
19 print(c.shape)
20
21 plt.scatter(x=range(0, dimension), y=a[0], color='blue')
22 plt.plot(range(0, dimension), a[0], color='blue')
23 plt.scatter(x=b, y=c, color='red')
24 plt.plot(range(0, dimension), vt_list, color='red')
25 plt.show()
```

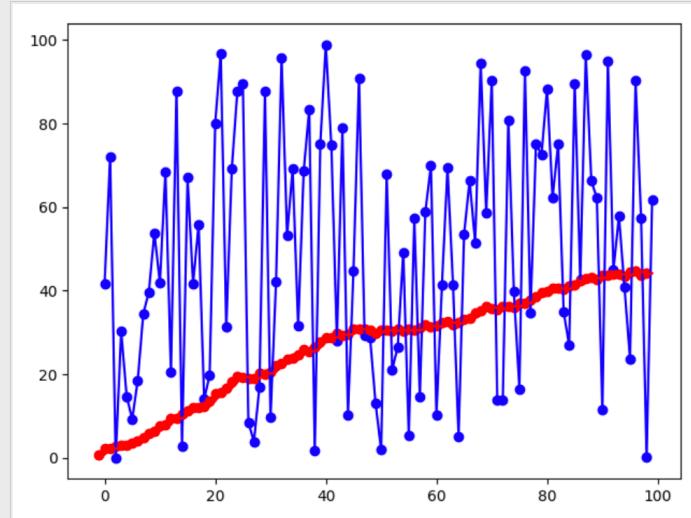
$\beta = 0.5$  时:



$\beta = 0.9$  时:



$\beta = 0.98$  时:



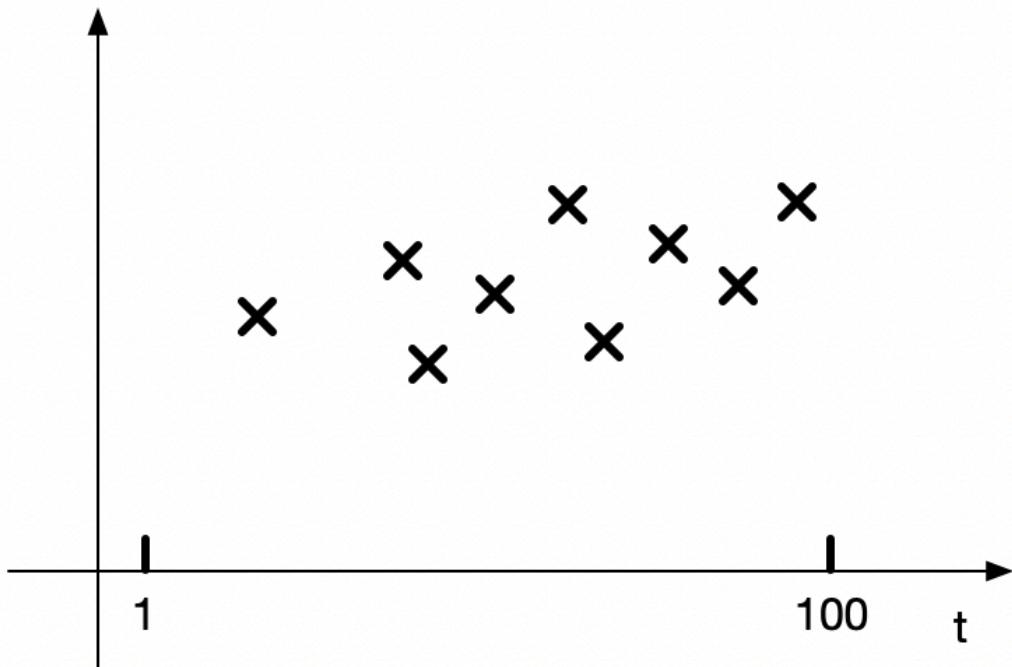
基本符合教科书中所描述的规律（注意看哦，图中红色曲线的起点较低哦，这个问题会在偏差修正中讲到）。

### 对于指数加权平均的理解

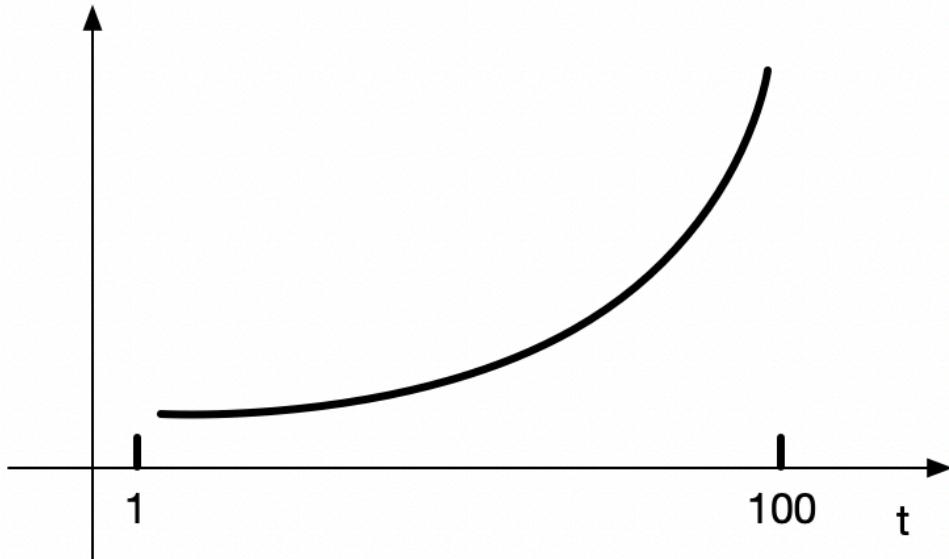
之前我对指数加权平均进行了推导总结，这里教科书给出了更为直观的解释。对于一组数据  $\theta_1, \theta_2, \dots, \theta_{100}$ ，和  $\beta = 0.9$  的指数加权平均  $v_{100}$  计算如下：

$$v_{100} = 0.1 * \theta_{100} + 0.1 * 0.9 * \theta_{99} + \dots + 0.1 * 0.9^{99} * \theta_1$$

我们将样本点和样本点系数分别做成两个图，x 轴从左到右均为第 1, …, 100 个样本，图例如下  
对于样本点：



样本系数（可以看作一个二次函数），样本角标越大，该值越大



最终的图样就可以看作是两图的数据对应相乘相加。

关于上节提到的，当  $\beta = 0.9$  时为 10 天的平均温度，这里作者的解释与上节相似，在这里简要说明。作者设  $\epsilon = 1 - \beta$ ，这里有一个公式  $(1 - \epsilon)^{1/\epsilon} \approx 1/e$ ，这里  $e$  为自然对数。不过用这个数代替平均值只是一个思考方向，并不是正式的数学证明（教程原话）。这里就不论数学证明，就把它作为平均值。

具体的实现过程，我就用上一节的实现来代替，这里跟作者讲述的思想一致：

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 dimension = 100
5 np.random.seed(1)
6 a = np.random.rand(1, dimension) * 100
7
8
9 vt = 0
10 vt_list = []
11 beta = 0.5
12 for i in range(dimension):
13     vt = vt * beta + (1. - beta) * a[0][i]
14     vt_list.append(vt)
15
16 b = np.array(range(-1, dimension-1))
17 c = np.array(vt_list)
18 print(b.shape)
19 print(c.shape)
20
21 plt.scatter(x=range(0, dimension), y=a[0], color='blue')
22 plt.plot(range(0, dimension), a[0], color='blue')
23 plt.scatter(x=b, y=c, color='red')
24 plt.plot(range(0, dimension), vt_list, color='red')
25 plt.show()

```

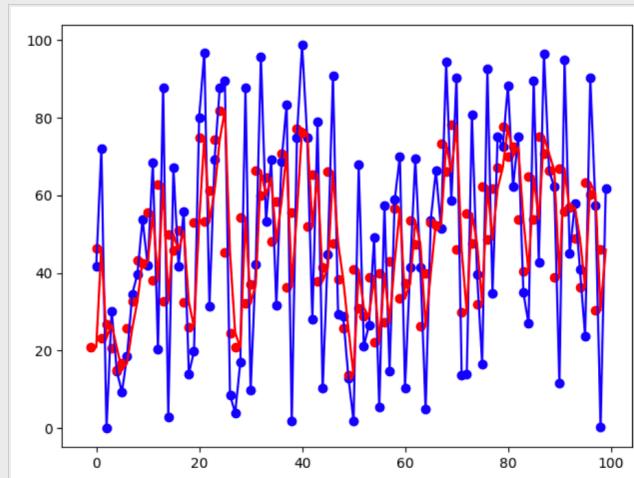
这里不用一般的平均值（如， $\frac{a+b}{2}$ ），用指数加权平均的好处在于：对于之前的 100 天气温的例子来说，若取 10 天的平均值，则需要找出这 10 天的值，做一个平均，存储成本和时间成本都会很高。如果用指数加权平均，那么就可以在每一次迭代中，只用一个变量记录下 10 天的平均值，能够节省时间。

除了指数加权平均，下一节还会教授一个内容叫做偏差修正，然后你就可以构建更好的优化算法，而不是简单直接的梯度下降算法。

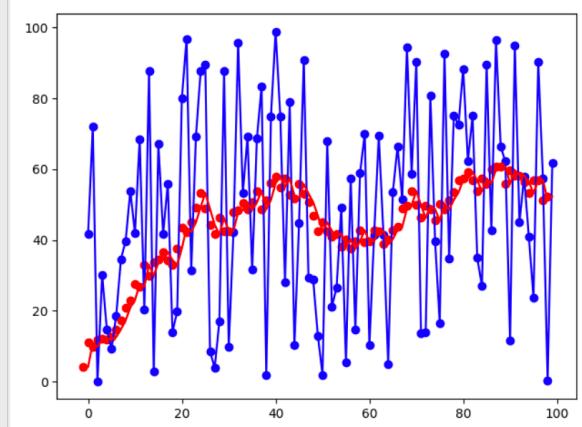
### 偏差修正 (bias correction)

如果你观察的足够仔细，就会发现在 指数加权平均 (exponentially weighted averages, EWA) 这里节中，我根据公式编写的代码，其执行结果与教程中给出的图有所偏差，体现在图像在起点数据值较低，如下图所示：

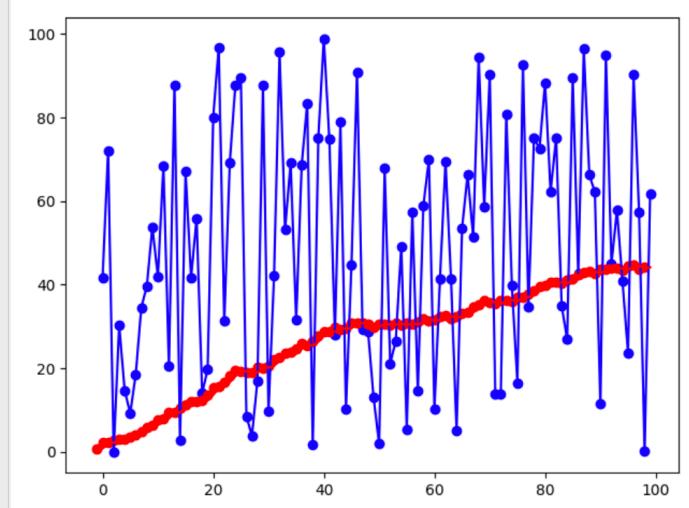
$\beta = 0.5$  时：



$\beta = 0.9$  时：



$\beta = 0.98$  时：



这里表现最明显的就是当  $\beta = 0.98$  时，教程也做了一下推导：

$$\begin{aligned}
 v_0 &= 0 \\
 v_1 &= 0.98 * v_0 + 0.02 * \theta_1 \\
 v_2 &= 0.98 * v_1 + 0.02 * \theta_2 \\
 &= 0.98 * (0.98 * v_0 + 0.02 * \theta_1) + 0.02\theta_2 \\
 &= 0.0196 * \theta_1 + 0.02 * \theta_2
 \end{aligned}$$

可以看出，不论是  $v_1$  还是  $v_2$  其预测值均比原先的  $\theta_1$  和  $\theta_2$  小很多，这也就不难理解为什么图像的起点这么低了。

一个有效办法是，修改预测值，使用  $v_t = 1/(1 - \beta^t)$ ，可以看到，当  $t = 1$  时，分子很小，得到的  $v_t$  也会比原先大很多，这使得预测变得更为准确。当  $t$  比较大时，分子比较大，这新的  $v_t$  与原先的  $v_t$  值比较接近。在原先的图像中，中后部分的曲线其实是比较吻合平均值数据的，因此几乎不需要做修正。所以该公式能够更好的使指数加权平均预测平均值。我自己也写了代码，实现如下：

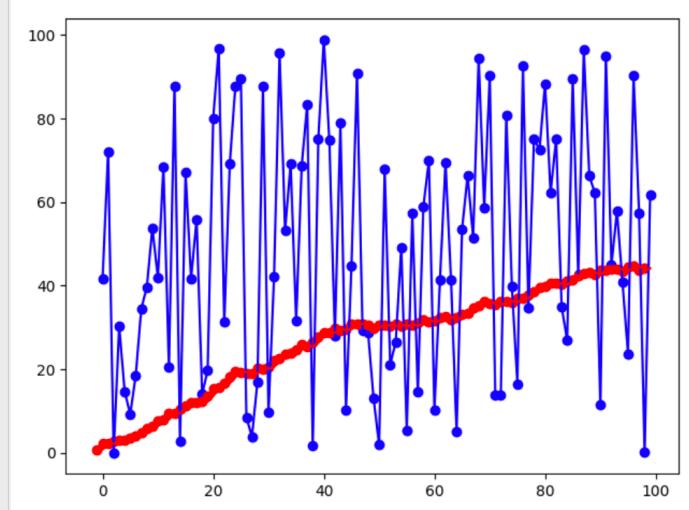
```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 dimension = 100
5 np.random.seed(1)
6 a = np.random.rand(1, dimension) * 100
7
8
9 vt = 0
10 vt_list = []
11 beta = 0.98
12 for i in range(1, dimension+1):
13     vt = vt * beta + (1. - beta) * a[0][i-1]
14     # vt_list.append(vt)
15     vt_list.append(vt / (1-beta ** i))
16
17 b = np.array(range(-1, dimension-1))
18 c = np.array(vt_list)
19 print(b.shape)

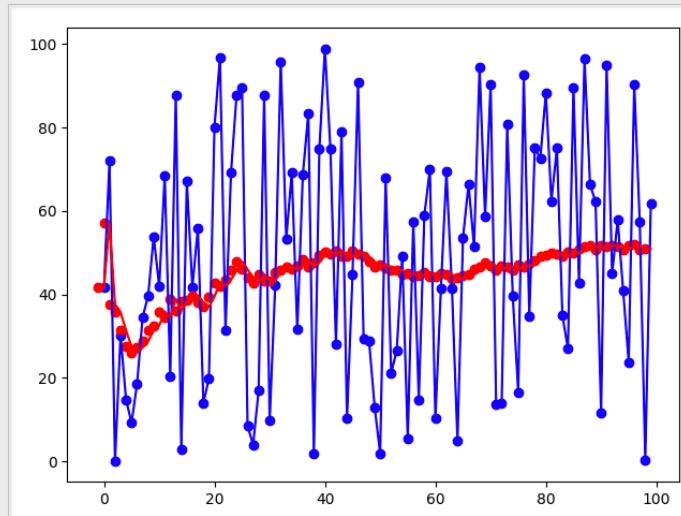
```

```
20 print(c.shape)
21
22 plt.scatter(x=range(0, dimension), y=a[0], color='blue')
23 plt.plot(range(0, dimension), a[0], color='blue')
24 plt.scatter(x=b, y=c, color='red')
25 plt.plot(range(0, dimension), vt_list, color='red')
26 plt.show()
```

未启用偏差修正



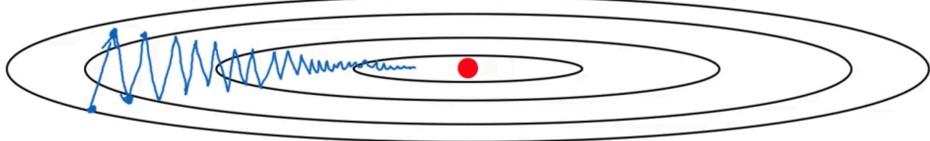
启用偏差修正



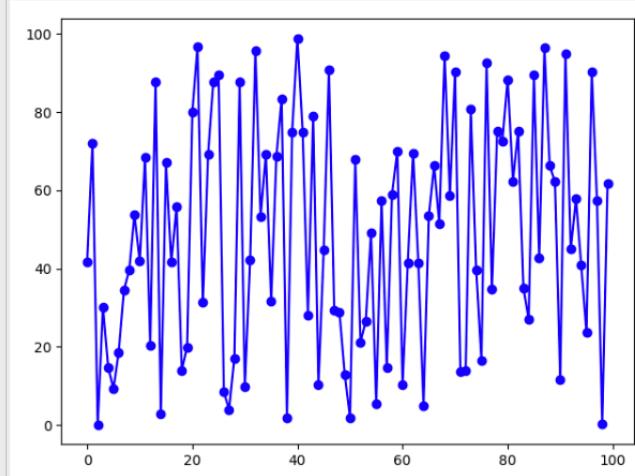
有了这两个算法，我们就可以构建优化算法了。

### 动量梯度下降法 (Momentum gradient descent)

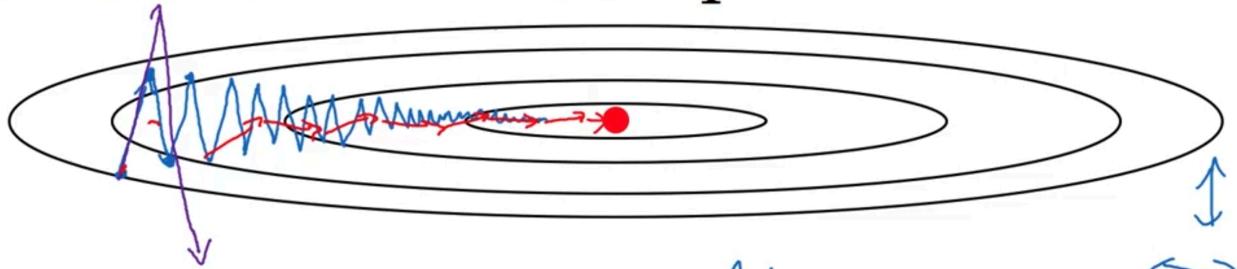
为了引出动量梯度下降法，我们先看一个关于 batch 或 mini-batch 梯度下降的图：



观察一下，会发现该梯度下降轨迹比较曲折形似下图：



嗯，如果你回忆一下就会想到，上节提到的指数加权平均能够使得这样直上直下的折线变得比较平坦，作用域梯度下降上时，会提升梯度下降，图例如下：



上图的红线，就是对梯度下降应用了指数加权平均值的梯度下降轨迹。

联系之前的知识，我们发现，影响梯度下降轨迹的因素主要是参数的导数  $dw$  和  $db$ ，因此我们需要对这两个导数进行指数加权平均，过程如下：

$$\begin{aligned}V_{dw} &= \beta * V_{dw} + (1 - \beta) * dw \\V_{db} &= \beta * V_{db} + (1 - \beta) * db \\w &= w - \alpha * V_{dw} \\b &= b - \alpha * V_{db}\end{aligned}$$

对于这里的  $\beta$  取值，教科书中认为  $\beta = 0.9$  具有很好的鲁棒性。公式  $\beta * V_{dw}$  中的  $V_{dw}$  按理说应该是前一个  $V_{dw}$  值，由于这里没有指明层数，因此教科书中没有写明，其他文献中可能会用下面这个公式：

$$\begin{aligned}V_{dw} &= \beta * V_{dw} + dw \\V_{db} &= \beta * V_{db} + db\end{aligned}$$

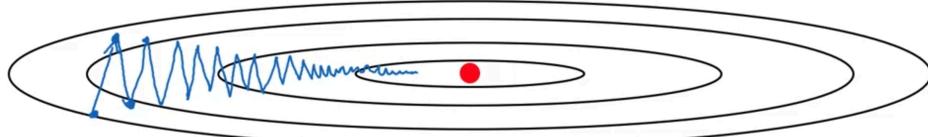
这个公式的效果也不错，如果应用这个公式，那么就需要对学习率进行一定的调整，我比较赞同教科书中使用的方法。

实际上，我们在使用这个方法进行梯度下降时，是不进行偏差调整的。这是因为训练 10 次之后，指数加权平均处理后的预测值已经接近平均值了，没必要进行偏差调整了。

除了 momentum 梯度下降，仍然有很多方法可以优化算法，我们在下一节内容会讲到。

## RMSprop (Root Mean Square prop)

让我们再来观察这个图

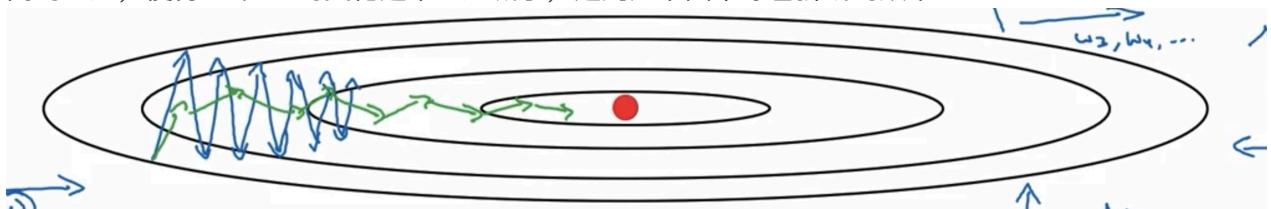


我们从导数的方向去考虑这个图像，假设在这个图中的垂直方向坐标轴为  $b$ ，水平方向坐标轴为  $w$ ，可以看出，这个折线在垂直方向的导数比水平方向的导数要大（这里请忽略尖点不可导的问题），这将使得  $w$  的变化速率比  $b$  的变化速率要低，导致折线上下摆动幅度很大，梯度下降效率低。RMSprop 算法用下面的思路解决这个问题：

$$\begin{aligned} S_{dw} &= \beta_2 * S_{dw} + (1 - \beta_2) * (dw)^2 \\ S_{db} &= \beta_2 * S_{db} + (1 - \beta_2) * (db)^2 \\ w &= w - \alpha * \frac{dw}{\sqrt{S_{dw}}} \\ b &= b - \alpha * \frac{db}{\sqrt{S_{db}}} \end{aligned}$$

上述公式中的  $\beta_2$  与之前 momentum 中的  $\beta$  含义是一样的，这里写作  $\beta_2$  是因为下一节要结合这两个方法，这样标记用于区分。

下面来了解一下它的原理。根据之前的假设， $dw$  是要比  $db$  小的。根据上面的公式得到的  $S_{dw}$  也是要比  $S_{db}$  小的。那么在更新  $w$  和  $b$  时， $1/\sqrt{S_{dw}}$  要比  $1/\sqrt{S_{db}}$  大，用它们分别就可以减小  $w$  和  $b$  之间的差距，使得  $w$  和  $b$  的变化速率差距减小，达到如下图中绿色折线的效果：



需要注意的是，上述这个公式可能会出现开根后近似为 0 的情况，所以一般我们在后面加上一个  $\epsilon$ ，一般取  $1 * 10^{-8}$ ，公式变成如下形式：

$$\begin{aligned} S_{dw} &= \beta_2 * S_{dw} + (1 - \beta_2) * (dw)^2 \\ S_{db} &= \beta_2 * S_{db} + (1 - \beta_2) * (db)^2 \\ w &= w - \alpha * \frac{dw}{\sqrt{S_{dw} + \epsilon}} \\ b &= b - \alpha * \frac{db}{\sqrt{S_{db} + \epsilon}} \end{aligned}$$

这样能够使得计算变得稳定。当然，这个方法允许你使用较大的学习率来提高梯度下降速率。下一节我们会将 momentum 方法和 RMSprop 方法放在一起使用。

## Adam (Adaptive Moment Estimation or authorization algorithm 权威算法) 自适应矩阵估计

Adam 算法是继 momentum 算法被广泛接受的深度学习算法，它实际上是 momentum 和 RMSprop 的结合，我们直接来看一下它的计算过程。

initialize :  $V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

compute :  $dw, db$  with mini - batch

$$V_{dw} = \beta_1 * V_{dw} + (1 - \beta_1) * dw$$

$$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) * db$$

$$S_{dw} = \beta_2 * S_{dw} + (1 - \beta_2) * (dw)^2$$

$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$$

$$V_{dw}^{corrected} = \frac{V_{dw}}{1 - \beta_1^t}$$

$$V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1 - \beta_2^t}$$

$$S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t}$$

$$w = w - \alpha * \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}$$

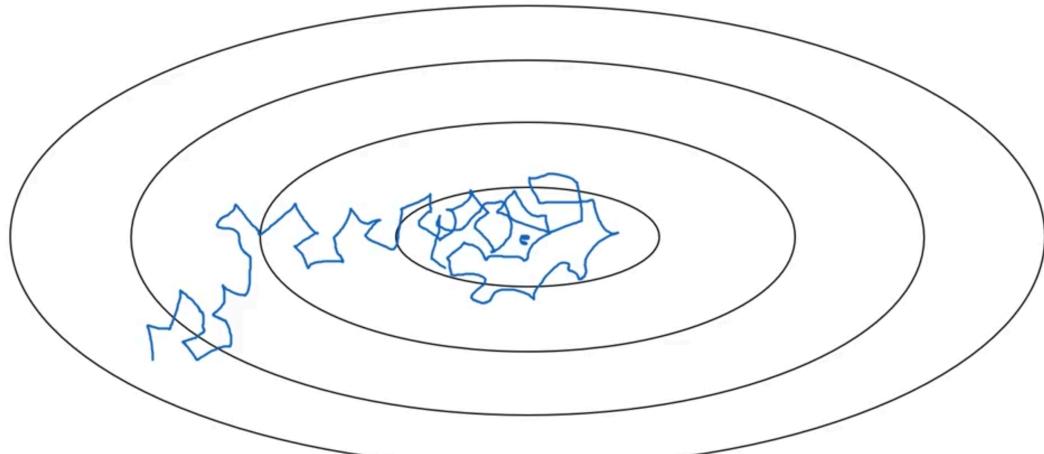
$$b = b - \alpha * \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$

上述被初始化的参数，值非零的是均为官方推荐的值，学习率需要自己另外调整， $\epsilon$  是一个可有可无的值，对结果影响不是很大。

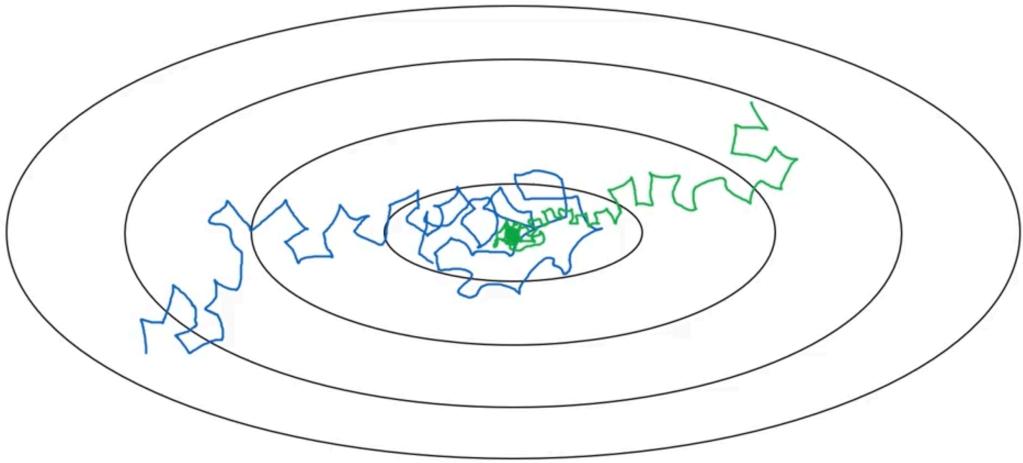
Adam (adaptive moment estimate) 这个名字的由来跟  $\beta_1$  和  $\beta_2$ ，他们分别被称作第一矩 (first moment) 和第二矩 (second moment)。下节内容我们学习学习率衰减，用来调整超参数  $\alpha$ 。

## 学习率衰减

还记得吗，我们在学习 mini-batch 的时候，如果没有调整到合适的 size，那么梯度下降最终不会收敛到最小值那个点，而是围绕着周围转圈圈。如下图所示：



一个让梯度下降最终结果很接近最小值的策略是降低学习率。如果学习率下降，意味着参数  $w$  和  $b$  的更新速度变慢，这会使得梯度下降的偏离程度变小，最终非常接近最小值。如下图绿线所示



这种方法被称为学习率衰减，有以下这个公式：

$$\alpha = \frac{1}{1 + decay\_rate * epoch\_num} * \alpha_0$$

公式中  $decay\_rate$  表示学习率下降参数， $epoch\_num$  表示几代（一代表第一次完整遍历 mini-batch 的所有训练集，二代表第二次，以此类推）。 $\alpha_0$  表示学习率初始值。可以带入几个值自己尝试计算一下，该公式会使得学习率随着  $epoch\_num$  的增加而减小。

除了这个公式，还有常用的几个公式：

$$\alpha = 0.95^{decay\_rate} * \alpha_0 \quad (1)$$

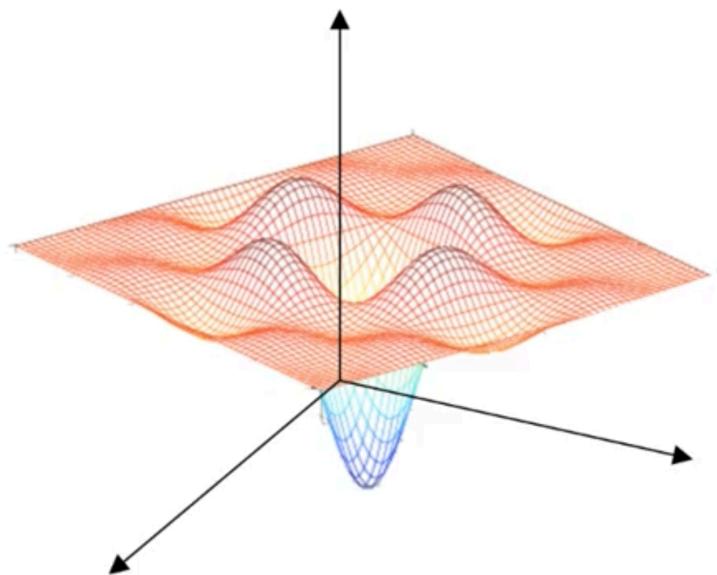
$$\alpha = \frac{k}{\sqrt{epoch\_num}} * \alpha_0 \quad (2)$$

$$\alpha = \frac{k}{\sqrt{t}} * \alpha_0 \quad (3)$$

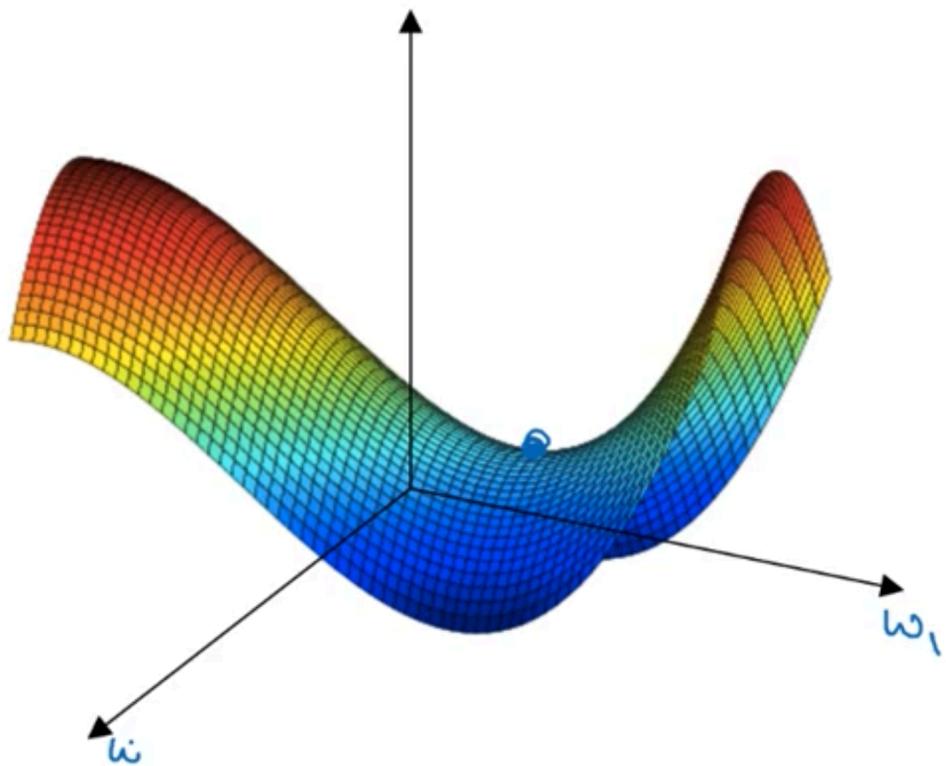
公式 (3) 中的  $t$  与之前举例中的天数含义相同。除了上述的公式之外，还有一种离散调整学习率的方法，为每隔一段时间调整一次学习率。当然也可以手动调整学习率。

## 局部最优和鞍点

这一节内容讲解了深度学习历史中出现的一个错误理解。以前，人们在进行梯度下降的时候，认为梯度为 0 的点为局部最优点，提到梯度下降，他们脑海里的图像是下图这样的：

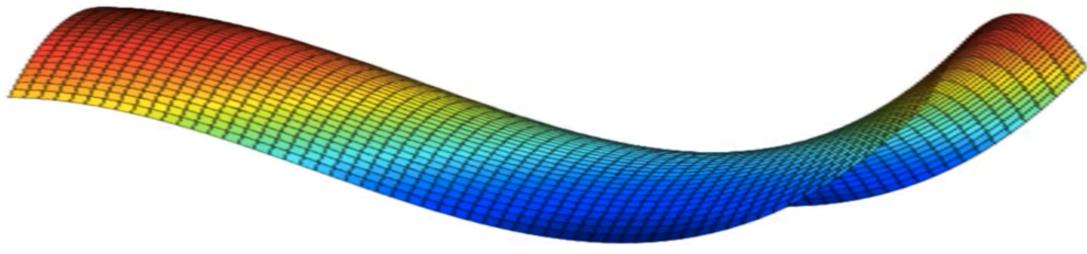


人们担心梯度下降算法会卡在橙色平面的最低点（局部最优），而不是到达蓝色最底部处（全局最优）。而现在，人们发现，在数据维度很高的情况下，例如：20000 个维度，要想使得局部最优出现，就要使得局部最优点在 20000 个维度中均为最小值（20000 阶导数均为 0），这个概率显然是很小的。人们发现，在高维情况下，梯度为 0 的点往往呈现出马鞍的形状，一个维度上是最低点，另一个维度上是最高等点，例如下图：



因此，在训练过程中，很难会出现卡在某个点不动弹的情况。上图如果起点在做上红黑色的地方，经过梯度下降，它会一步一步走到中间的淡蓝色地方（被称为鞍点），然后朝着蓝紫色的地方继续下降。

算法效率令人担心的地方其实是中间一段较为平坦的地方，如下图：



从黄色到蓝色区域这一段比较平坦，意味着梯度值比较小，梯度下降缓慢，并且会出现左右扰动的情况（例如红色区域可能在梯度下降的时候朝向黄色区域的地方前进，然后又朝着绿色区域前进），所以我们可以使用之前学习的 momentum 算法、RMSprop 算法或者 Adam 算法来梯度下降的效率。关于局部最优和鞍点更详细的解释，可以参考这篇文章 [局部最优和鞍点区分--CSDN](#)。

## 课后编程练习 1

本编程练习内容针对优化算法进行练习，本练习作者给的源代码在 Week2\_编程练习/Optimization\_methods/codes 下，数据集在 Week2\_编程练习/Optimization\_methods/datasets 下。

首先，作者构建了一个更新参数的函数，如下段代码所示，比较简单：

```

1 def update_parameters_with_gd(parameters, grads, learning_rate):
2     L = len(parameters) // 2
3     for i in range(L):
4         parameters["W" + str(i+1)] = parameters["W" + str(i+1)] -
5             learning_rate * grads['dW' + str(i+1)]
6         parameters["b" + str(i+1)] = parameters["b" + str(i+1)] -
7             learning_rate * grads['db' + str(i+1)]
8     return parameters

```

测试代码以及测试结果如下：

```

1 parameters, grads, learning_rate = update_parameters_with_gd_test_case()
2 parameters = update_parameters_with_gd(parameters, grads, learning_rate)
3 print("W1 = " + str(parameters["W1"]))
4 print("b1 = " + str(parameters["b1"]))
5 print("W2 = " + str(parameters["W2"]))
6 print("b2 = " + str(parameters["b2"]))
7 ##### result #####
8 W1 = [[ 1.63535156 -0.62320365 -0.53718766][-1.07799357  0.85639907
-2.29470142]]
9 b1 = [[ 1.74604067][-0.75184921]]
10 W2 = [[ 0.32171798 -0.25467393  1.46902454][-2.05617317 -0.31554548
-0.3756023 ][ 1.1404819  -1.09976462 -0.1612551 ]]
11 b2 = [[-0.88020257][ 0.02561572][ 0.57539477]]

```

该算法 (Batch gradient decent) 的一种变体叫做随机梯度下降 (stochastic gradient decent) , 之前的内容也介绍过这两种算法以及 mini-batch 算法的优势和劣势了, 我们不多介绍这两种算法了, 直接实现 mini-batch 梯度下降算法。

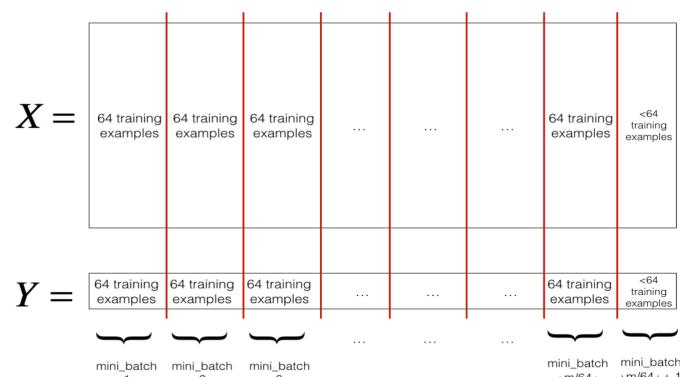
mini-batch 梯度下降算法氛围两步:

1. shuffle (随机组合) : 随机组合训练集 X 和标签 Y, 使得最后划分 mini-batch 的时候, 能够均匀分布每一个训练样本到每一个 mini-batch, 但 X 与 Y 要一一对应, 图例如下:

$$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \dots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \dots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix} \quad Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$

$$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \dots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \dots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix} \quad Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$

2. partition (划分) : 将随机组合后的训练集划分成多个 mini-batch, 例如每个 mini-batch 中有 64 个训练样本。但是不一定每次划分都能刚好使得每个 mini-batch 都有 64 个训练样本, 最后一个总会小于等于 64。但这个并无大碍。划分图例如下所示:



接下来我们就来实现这个算法。代码如下

```

1 # GRADED FUNCTION: random_mini_batches
2
3 def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
4     np.random.seed(seed)                      # To make your "random" minibatches
5     the same as ours
6     m = X.shape[1]                            # number of training examples
7     mini_batches = []
8
9     # Step 1: Shuffle (X, Y)
10    permutation = list(np.random.permutation(m)) #
11    np.random.permutation(m) 函数给出一个随机序列, 根据这个序列实现训练集的随机组合。
12    shuffled_X = X[:, permutation]
13    shuffled_Y = Y[:, permutation].reshape((1,m))
14
15    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
16    num_complete_minibatches = math.floor(m/mini_batch_size) # number of
17    mini batches of size mini_batch_size in your partitionning

```

```

15     for k in range(0, num_complete_minibatches):
16         ### START CODE HERE ### (approx. 2 lines)
17         mini_batch_X = shuffled_X[:, k * mini_batch_size:
18                                     (k+1) * mini_batch_size]
19         mini_batch_Y = shuffled_Y[:, k * mini_batch_size:
20                                     (k+1) * mini_batch_size]
21         ### END CODE HERE ###
22         mini_batch = (mini_batch_X, mini_batch_Y)
23         mini_batches.append(mini_batch)
24
25     # Handling the end case (last mini-batch < mini_batch_size)
26     if m % mini_batch_size != 0:
27         ### START CODE HERE ### (approx. 2 lines)
28         mini_batch_X = shuffled_X[:, (num_complete_minibatches) * mini_batch_size:]
29         mini_batch_Y = shuffled_Y[:, (num_complete_minibatches) * mini_batch_size:]
30         ### END CODE HERE ###
31         mini_batch = (mini_batch_X, mini_batch_Y)
32         mini_batches.append(mini_batch)
33
34
35     return mini_batches

```

具体的算法实现思路在代码里面都有体现哦，可以仔细看看。测试代码和测试结果如下：

```

1 X_assess, Y_assess, mini_batch_size = random_mini_batches_test_case()
2 mini_batches = random_mini_batches(X_assess, Y_assess, mini_batch_size)
3
4 print ("shape of the 1st mini_batch_X: " + str(mini_batches[0][0].shape))
5 print ("shape of the 2nd mini_batch_X: " + str(mini_batches[1][0].shape))
6 print ("shape of the 3rd mini_batch_X: " + str(mini_batches[2][0].shape))
7 print ("shape of the 1st mini_batch_Y: " + str(mini_batches[0][1].shape))
8 print ("shape of the 2nd mini_batch_Y: " + str(mini_batches[1][1].shape))
9 print ("shape of the 3rd mini_batch_Y: " + str(mini_batches[2][1].shape))
10 print ("mini batch sanity check: " + str(mini_batches[0][0][0][0:3]))
11 ##### result #####
12 shape of the 1st mini_batch_X: (12288, 64)
13 shape of the 2nd mini_batch_X: (12288, 64)
14 shape of the 3rd mini_batch_X: (12288, 20)
15 shape of the 1st mini_batch_Y: (1, 64)
16 shape of the 2nd mini_batch_Y: (1, 64)
17 shape of the 3rd mini_batch_Y: (1, 20)
18 mini batch sanity check: [ 0.90085595 -0.7612069   0.2344157 ]

```

由于 mini-batch 算法使得梯度下降变得上下颠簸，所以我们需要使用 momentum 算法将梯度下降曲线变得比较平缓。由于已经在之前讲过 momentum 算法，因此这里就不再赘述了，我们从  $V_{dw}$  和  $V_{db}$  的初始化开始实现，我们的思路是将这些参数初始化为与参数 w 和 b 维度相同的 0 向量。参数的初始化如下段代码所示：

```

1 # GRADED FUNCTION: initialize_velocity
2
3 def initialize_velocity(parameters):
4     L = len(parameters) // 2 # number of layers in the neural networks
5     v = {}
6
7     # Initialize velocity
8     for l in range(L):
9         ### START CODE HERE ### (approx. 2 lines)
10        v[ "dW" + str(l+1)] = np.zeros((parameters[ 'W'+str(l+1)].shape[0],
parameters[ 'W'+str(l+1)].shape[1]))
11        v[ "db" + str(l+1)] = np.zeros((parameters[ 'b'+str(l+1)].shape[0],
parameters[ 'b'+str(l+1)].shape[1]))
12        ### END CODE HERE ###
13
14    return v

```

测试代码和结果如下：

```

1 parameters = initialize_velocity_test_case()
2 v = initialize_velocity(parameters)
3 print("v[ \"dW1\" ] = " + str(v[ "dW1"]))
4 print("v[ \"db1\" ] = " + str(v[ "db1"]))
5 print("v[ \"dW2\" ] = " + str(v[ "dW2"]))
6 print("v[ \"db2\" ] = " + str(v[ "db2"]))
7 ##### result #####
8 v[ "dW1" ] = [[0. 0. 0.] [0. 0. 0.]]
9 v[ "db1" ] = [[0.] [0.]]
10 v[ "dW2" ] = [[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
11 v[ "db2" ] = [[0.] [0.] [0.]]

```

接下来就是根据 momentum 给出的公式更新参数，公式如下：

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1 - \beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1 - \beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

代码如下：

```

1 # GRADED FUNCTION: update_parameters_with_momentum
2
3 def update_parameters_with_momentum(parameters, grads, v, beta,
learning_rate):
4     L = len(parameters) // 2 # number of layers in the neural networks
5     # Momentum update for each parameter
6     for l in range(L):
7         ### START CODE HERE ### (approx. 4 lines)
8         # compute velocities

```

```

9      v[ "dw" + str(l+1) ] = beta * v[ "dw" + str(l+1) ] + (1-beta) *
grads[ 'dw'+str(l+1) ]
10     v[ "db" + str(l+1) ] = beta * v[ "db" + str(l+1) ] + (1-beta) *
grads[ 'db'+str(l+1) ]
11     # update parameters
12     parameters[ "W" + str(l+1) ] = parameters[ "W" + str(l+1) ] -
learning_rate * v[ 'dw' + str(l+1) ]
13     parameters[ "b" + str(l+1) ] = parameters[ "b" + str(l+1) ] -
learning_rate * v[ 'db' + str(l+1) ]
14     #### END CODE HERE ####
15
return parameters, v

```

测试代码和测试结果如下：

```

1 parameters, grads, v = update_parameters_with_momentum_test_case()
2 parameters, v = update_parameters_with_momentum(parameters, grads, v, beta
= 0.9, learning_rate = 0.01)
3 print("W1 = " + str(parameters[ "W1" ]))
4 print("b1 = " + str(parameters[ "b1" ]))
5 print("W2 = " + str(parameters[ "W2" ]))
6 print("b2 = " + str(parameters[ "b2" ]))
7 print("v[ \"dw1\" ] = " + str(v[ "dw1" ]))
8 print("v[ \"db1\" ] = " + str(v[ "db1" ]))
9 print("v[ \"dw2\" ] = " + str(v[ "dw2" ]))
10 print("v[ \"db2\" ] = " + str(v[ "db2" ]))
11 ##### result #####
12 W1 = [[ 1.62544598 -0.61290114 -0.52907334]
13 [-1.07347112  0.86450677 -2.30085497]]
14 b1 = [[ 1.74493465]
15 [-0.76027113]]
16 W2 = [[ 0.31930698 -0.24990073  1.4627996 ]
17 [-2.05974396 -0.32173003 -0.38320915]
18 [ 1.13444069 -1.0998786 -0.1713109 ]]
19 b2 = [[-0.87809283]
20 [ 0.04055394]
21 [ 0.58207317]]
22 v[ "dw1" ] = [[-0.11006192  0.11447237  0.09015907]
23 [ 0.05024943  0.09008559 -0.06837279]]
24 v[ "db1" ] = [[-0.01228902]
25 [-0.09357694]]
26 v[ "dw2" ] = [[-0.02678881  0.05303555 -0.06916608]
27 [-0.03967535 -0.06871727 -0.08452056]
28 [-0.06712461 -0.00126646 -0.11173103]]
29 v[ "db2" ] = [[0.02344157]
30 [ 0.16598022]
31 [ 0.07420442]]

```

结合两者的优点的算法就是 Adam 算法，下面我们来实现这个算法。

我们先初始化算法用到的参数，同样的，我们把它们初始化为与 w 和 b 维度相同的零向量，代码

如下：

```
1 def initialize_adam(parameters) :
2     L = len(parameters) // 2 # number of layers in the neural networks
3     v = {}
4     s = {}
5
6     # Initialize v, s. Input: "parameters". Outputs: "v, s".
7     for l in range(L):
8         ### START CODE HERE ### (approx. 4 lines)
9         v[ "dw" + str(l+1)] = np.zeros((parameters[ 'W'+str(l+1)].shape[0],
10                                         parameters[ 'W'+str(l+1)].shape[1]))
11        v[ "db" + str(l+1)] = np.zeros((parameters[ 'b'+str(l+1)].shape[0],
12                                         parameters[ 'b'+str(l+1)].shape[1]))
13        s[ "dw" + str(l+1)] = np.zeros((parameters[ 'W'+str(l+1)].shape[0],
14                                         parameters[ 'W'+str(l+1)].shape[1]))
15        s[ "db" + str(l+1)] = np.zeros((parameters[ 'b'+str(l+1)].shape[0],
16                                         parameters[ 'b'+str(l+1)].shape[1]))
17
18     ### END CODE HERE ###
19
20
21     return v, s
```

测试代码和测试结果如下：

```
1 parameters = initialize_adam_test_case()
2 v, s = initialize_adam(parameters)
3 print("v[\"dw1\"] = " + str(v[ "dw1"]))
4 print("v[\"db1\"] = " + str(v[ "db1"]))
5 print("v[\"dw2\"] = " + str(v[ "dw2"]))
6 print("v[\"db2\"] = " + str(v[ "db2"]))
7 print("s[\"dw1\"] = " + str(s[ "dw1"]))
8 print("s[\"db1\"] = " + str(s[ "db1"]))
9 print("s[\"dw2\"] = " + str(s[ "dw2"]))
10 print("s[\"db2\"] = " + str(s[ "db2"]))
11 ##### result #####
12 v[ "dw1"] = [[0. 0. 0.] [0. 0. 0.]]
13 v[ "db1"] = [[0.] [0.]]
14 v[ "dw2"] = [[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
15 v[ "db2"] = [[0.] [0.] [0.]]
16 s[ "dw1"] = [[0. 0. 0.] [0. 0. 0.]]
17 s[ "db1"] = [[0.] [0.]]
18 s[ "dw2"] = [[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
19 s[ "db2"] = [[0.] [0.][0.]]
```

接着，我们根据下面的公式来更新参数：

$$\left\{ \begin{array}{l} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1 - \beta_1) \frac{\partial J}{\partial W^{[l]}} \\ v_{W^{[l]}}^{corrected} = \frac{v_{W^{[l]}}}{1 - (\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1 - \beta_2) \left( \frac{\partial J}{\partial W^{[l]}} \right)^2 \\ s_{W^{[l]}}^{corrected} = \frac{s_{W^{[l]}}}{1 - (\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v_{W^{[l]}}^{corrected}}{\sqrt{s_{W^{[l]}}^{corrected} + \epsilon}} \end{array} \right.$$

代码实现为：

```
1 # GRADED FUNCTION: update_parameters_with_adam
2
3 def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate
4 = 0.01,
5
6     beta1 = 0.9, beta2 = 0.999, epsilon = 1e-
7 8):
8
9     L = len(parameters) // 2
10    # number of layers in the
11    neural networks
12    v_corrected = {} # Initializing first moment
13    estimate, python dictionary
14    s_corrected = {} # Initializing second moment
15    estimate, python dictionary
16
17    # Perform Adam update on all parameters
18    for l in range(L):
19        # Moving average of the gradients. Inputs: "v, grads, beta1".
20        Output: "v".
21        ### START CODE HERE ### (approx. 2 lines)
22        v[ "dw" + str(l+1) ] = beta1 * v[ "dw" + str(l+1) ] + (1-beta1) *
23        grads[ 'dw'+str(l+1) ]
24        v[ "db" + str(l+1) ] = beta1 * v[ "db" + str(l+1) ] + (1-beta1) *
25        grads[ 'db'+str(l+1) ]
26        ### END CODE HERE ###
27
28        # Compute bias-corrected first moment estimate. Inputs: "v, beta1,
29        t". Output: "v_corrected".
30        ### START CODE HERE ### (approx. 2 lines)
31        v_corrected[ "dw" + str(l+1) ] = v[ "dw" + str(l+1) ] / (1 -
32        np.power(beta1, t))
33        v_corrected[ "db" + str(l+1) ] = v[ "db" + str(l+1) ] / (1 -
34        np.power(beta1, t))
35        ### END CODE HERE ###
36
37        # Moving average of the squared gradients. Inputs: "s, grads,
38        beta2". Output: "s".
39        ### START CODE HERE ### (approx. 2 lines)
```

```

26         s["dw" + str(l+1)] = beta2 * s["dw" + str(l+1)] + (1-beta2) *
27             (grads['dw'+str(l+1)] ** 2)
28         s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2) *
29             (grads['db'+str(l+1)] **2)
30             ### END CODE HERE ###
31
32             # Compute bias-corrected second raw moment estimate. Inputs: "s,
33             beta2, t". Output: "s_corrected".
34             ### START CODE HERE ### (approx. 2 lines)
35             s_corrected["dw" + str(l+1)] = s["dw" + str(l+1)] / (1 - beta2 **
36             t)
37             s_corrected["db" + str(l+1)] = s["db" + str(l+1)] / (1 - beta2 **
38             t)
39             ### END CODE HERE ###
40
41
42     return parameters, v, s

```

测试代码和测试数据如下：

```

1 parameters, grads, v, s = update_parameters_with_adam_test_case()
2 parameters, v, s = update_parameters_with_adam(parameters, grads, v, s, t
= 2)
3 print("W1 = " + str(parameters["W1"]))
4 print("b1 = " + str(parameters["b1"]))
5 print("W2 = " + str(parameters["W2"]))
6 print("b2 = " + str(parameters["b2"]))
7 print("v[\"dw1\"] = " + str(v["dw1"]))
8 print("v[\"db1\"] = " + str(v["db1"]))
9 print("v[\"dw2\"] = " + str(v["dw2"]))
10 print("v[\"db2\"] = " + str(v["db2"]))
11 print("s[\"dw1\"] = " + str(s["dw1"]))
12 print("s[\"db1\"] = " + str(s["db1"]))
13 print("s[\"dw2\"] = " + str(s["dw2"]))
14 print("s[\"db2\"] = " + str(s["db2"]))
15 ##### result #####
16 W1 = [[ 1.63178673 -0.61919778 -0.53561312] [-1.08040999  0.85796626
-2.29409733]]
17 b1 = [[ 1.75225313] [-0.75376553]]

```

```

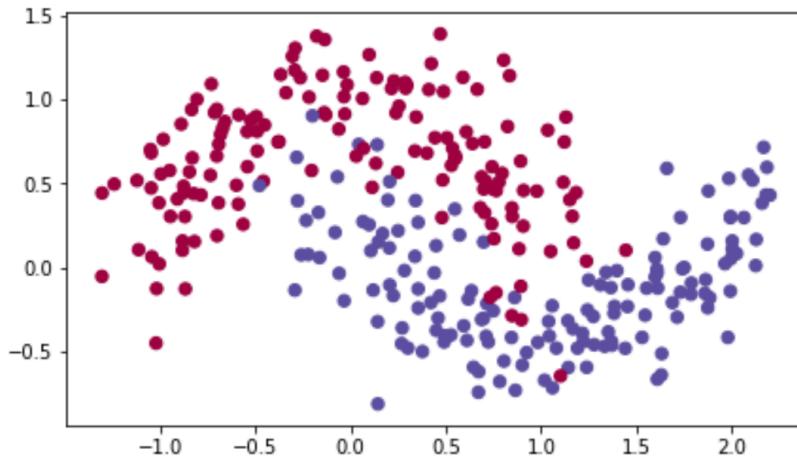
18 W2 = [[ 0.32648046 -0.25681174  1.46954931] [-2.05269934 -0.31497584
-0.37661299] [ 1.14121081 -1.09244991 -0.16498684]]
19 b2 = [[-0.88529979] [ 0.03477238] [ 0.57537385]]
20 v[ "dw1" ] = [[-0.11006192  0.11447237  0.09015907] [ 0.05024943  0.09008559
-0.06837279]]
21 v[ "db1" ] = [[-0.01228902] [-0.09357694]]
22 v[ "dw2" ] = [[-0.02678881  0.05303555 -0.06916608] [-0.03967535 -0.06871727
-0.08452056] [-0.06712461 -0.00126646 -0.11173103]]
23 v[ "db2" ] = [[0.02344157] [0.16598022] [0.07420442]]
24 s[ "dw1" ] = [[0.00121136  0.00131039  0.00081287] [0.0002525  0.00081154
0.00046748]]
25 s[ "db1" ] = [[1.51020075e-05] [8.75664434e-04]]
26 s[ "dw2" ] = [[7.17640232e-05 2.81276921e-04 4.78394595e-04] [1.57413361e-04
4.72206320e-04 7.14372576e-04] [4.50571368e-04 1.60392066e-07 1.24838242e-
03]]
27 s[ "db2" ] = [[5.49507194e-05] [2.75494327e-03] [5.50629536e-04]]

```

最后，我们来比较一下三个方法 (mini-batch, momentum, Adam) 吧：

数据集：

```
1 | train_X, train_Y = load_dataset()
```



model 函数：

```

1 | def model(X, Y, layers_dims, optimizer, learning_rate = 0.0007,
2 |             mini_batch_size = 64, beta = 0.9,
3 |                     beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, num_epochs = 10000,
4 |                     print_cost = True):
5 |     L = len(layers_dims)                                     # number of layers in the neural
networks
6 |     costs = []                                              # to keep track of the cost
7 |     t = 0                                                    # initializing the counter required
for Adam update
8 |     seed = 10                                                 # For grading purposes, so that your
"random" minibatches are the same as ours
9 |
# Initialize parameters

```

```

9     parameters = initialize_parameters(layers_dims)
10
11     # Initialize the optimizer
12     if optimizer == "gd":
13         pass # no initialization required for gradient descent
14     elif optimizer == "momentum":
15         v = initialize_velocity(parameters)
16     elif optimizer == "adam":
17         v, s = initialize_adam(parameters)
18
19     # Optimization loop
20     for i in range(num_epochs):
21
22         # Define the random minibatches. We increment the seed to
23         # reshuffle differently the dataset after each epoch
24         seed = seed + 1
25         minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
26
27         for minibatch in minibatches:
28
29             # Select a minibatch
30             (minibatch_X, minibatch_Y) = minibatch
31
32             # Forward propagation
33             a3, caches = forward_propagation(minibatch_X, parameters)
34
35             # Compute cost
36             cost = compute_cost(a3, minibatch_Y)
37
38             # Backward propagation
39             grads = backward_propagation(minibatch_X, minibatch_Y, caches)
40
41             # Update parameters
42             if optimizer == "gd":
43                 parameters = update_parameters_with_gd(parameters, grads,
44                 learning_rate)
45             elif optimizer == "momentum":
46                 parameters, v =
47                 update_parameters_with_momentum(parameters, grads, v, beta,
48                 learning_rate)
49             elif optimizer == "adam":
50                 t = t + 1 # Adam counter
51                 parameters, v, s = update_parameters_with_adam(parameters,
52                 grads, v, s,
53                 t,
54                 learning_rate, betal, beta2, epsilon)
55
56             # Print the cost every 1000 epoch
57             if print_cost and i % 1000 == 0:
58                 print ("Cost after epoch %i: %f" %(i, cost))

```

```

53     if print_cost and i % 100 == 0:
54         costs.append(cost)
55
56     # plot the cost
57     plt.plot(costs)
58     plt.ylabel('cost')
59     plt.xlabel('epochs (per 100)')
60     plt.title("Learning rate = " + str(learning_rate))
61     plt.show()
62
63 return parameters

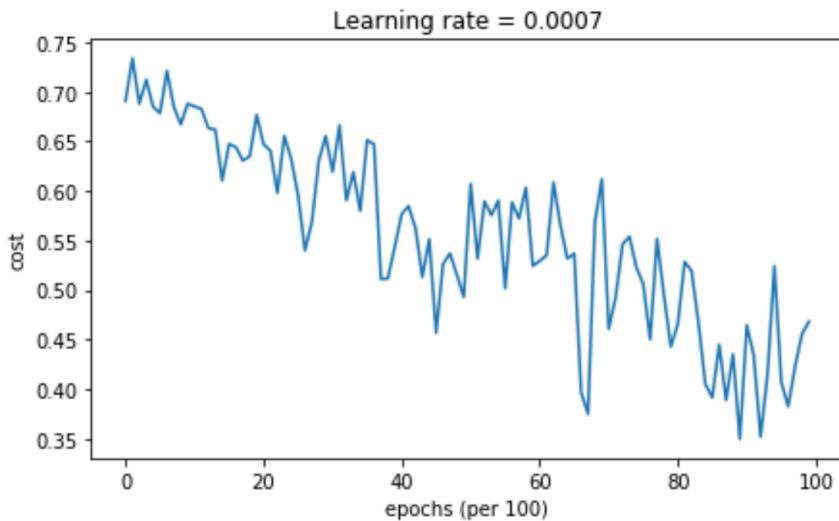
```

## 1. mini-batch

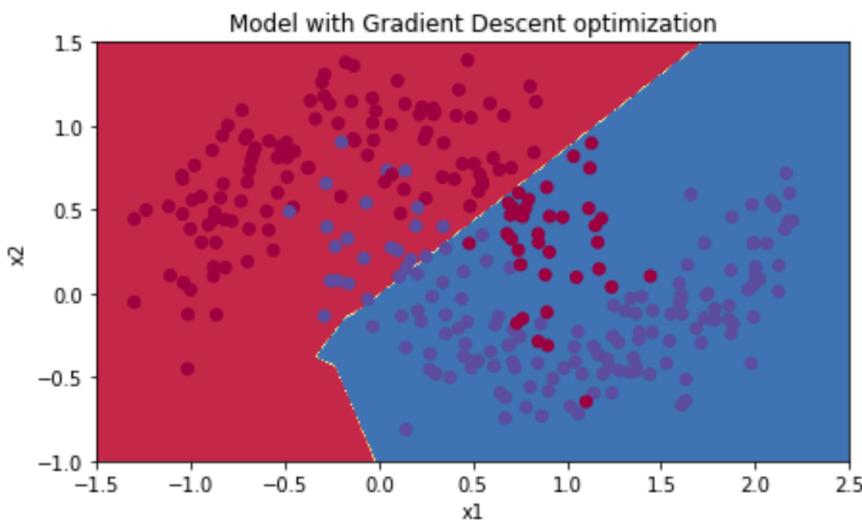
```

1 # train 3-layer model
2 layers_dims = [train_X.shape[0], 5, 2, 1]
3 parameters = model(train_X, train_Y, layers_dims, optimizer = "gd")
4
5 # Predict
6 predictions = predict(train_X, train_Y, parameters)
7
8 # Plot decision boundary
9 plt.title("Model with Gradient Descent optimization")
10 axes = plt.gca()
11 axes.set_xlim([-1.5,2.5])
12 axes.set_ylim([-1,1.5])
13 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
14 train_Y)
14 ##### result #####
15 Cost after epoch 0: 0.690736
16 Cost after epoch 1000: 0.685273
17 Cost after epoch 2000: 0.647072
18 Cost after epoch 3000: 0.619525
19 Cost after epoch 4000: 0.576584
20 Cost after epoch 5000: 0.607243
21 Cost after epoch 6000: 0.529403
22 Cost after epoch 7000: 0.460768
23 Cost after epoch 8000: 0.465586
24 Cost after epoch 9000: 0.464518

```



Accuracy: 0.7966666666666666



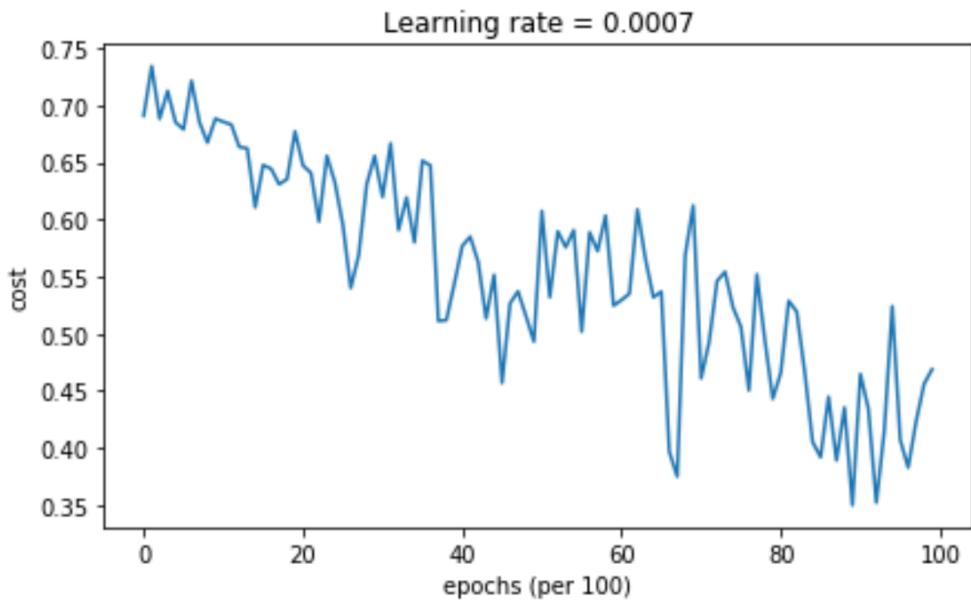
## 2. momentum

```

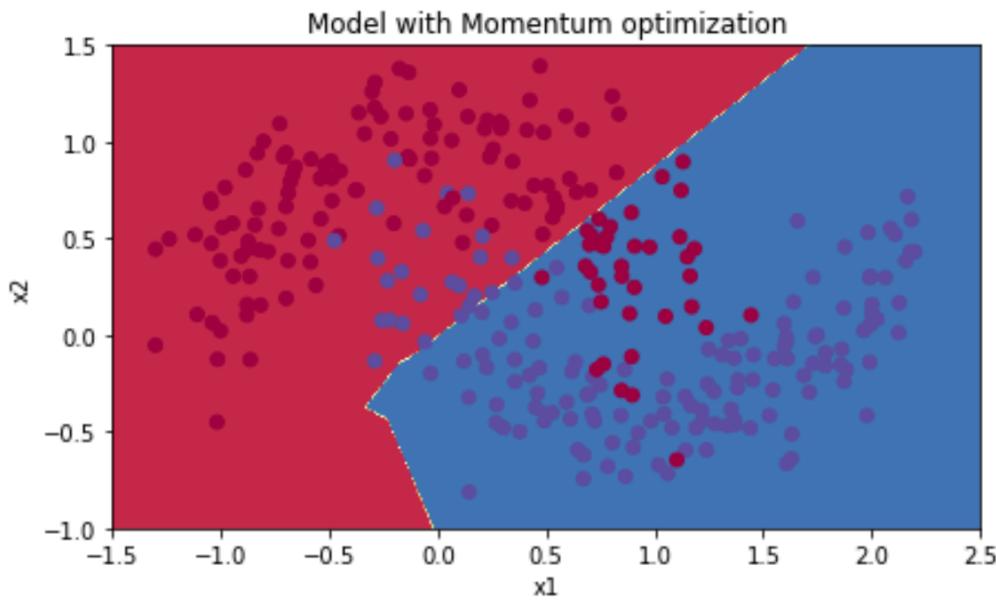
1 # train 3-layer model
2 layers_dims = [train_X.shape[0], 5, 2, 1]
3 parameters = model(train_X, train_Y, layers_dims, beta = 0.9, optimizer =
4 "momentum")
5
6 # Predict
7 predictions = predict(train_X, train_Y, parameters)
8
9 # Plot decision boundary
10 plt.title("Model with Momentum optimization")
11 axes = plt.gca()
12 axes.set_xlim([-1.5,2.5])
13 axes.set_ylim([-1,1.5])
14 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
15 train_Y)
16 ##### result #####
17 Cost after epoch 0: 0.690741
18 Cost after epoch 1000: 0.685341

```

```
17 Cost after epoch 2000: 0.647145
18 Cost after epoch 3000: 0.619594
19 Cost after epoch 4000: 0.576665
20 Cost after epoch 5000: 0.607324
21 Cost after epoch 6000: 0.529476
22 Cost after epoch 7000: 0.460936
23 Cost after epoch 8000: 0.465780
24 Cost after epoch 9000: 0.464740
```



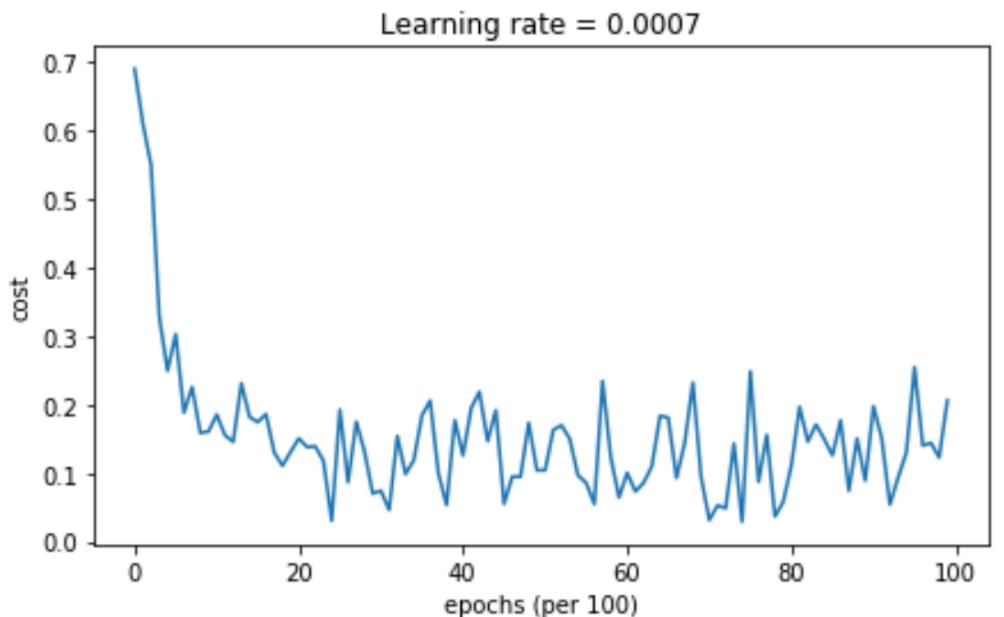
Accuracy: 0.7966666666666666



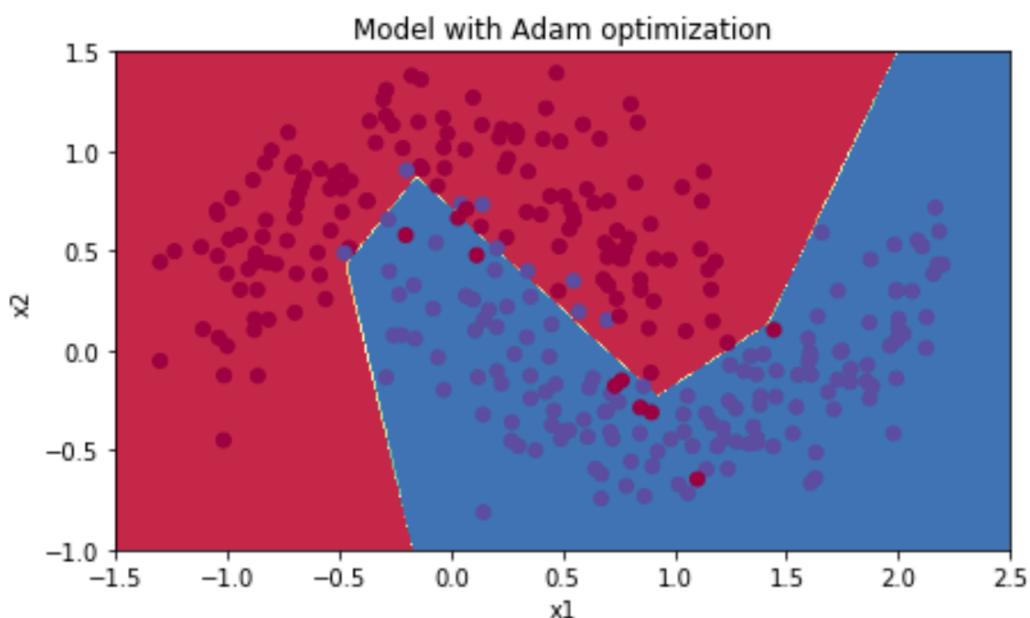
### 3. Adam

```
1 # train 3-layer model
2 layers_dims = [train_X.shape[0], 5, 2, 1]
3 parameters = model(train_X, train_Y, layers_dims, optimizer = "adam")
4
5 # Predict
```

```
6 predictions = predict(train_X, train_Y, parameters)
7
8 # Plot decision boundary
9 plt.title("Model with Adam optimization")
10 axes = plt.gca()
11 axes.set_xlim([-1.5,2.5])
12 axes.set_ylim([-1,1.5])
13 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X,
14 train_Y)
14 ##### result #####
15 Cost after epoch 0: 0.690552
16 Cost after epoch 1000: 0.185567
17 Cost after epoch 2000: 0.150852
18 Cost after epoch 3000: 0.074454
19 Cost after epoch 4000: 0.125936
20 Cost after epoch 5000: 0.104235
21 Cost after epoch 6000: 0.100552
22 Cost after epoch 7000: 0.031601
23 Cost after epoch 8000: 0.111709
24 Cost after epoch 9000: 0.197648
```



Accuracy: 0.94



可以看出 Adam 效果极佳!

## Week 3 超参数调试、Batch 正则化和程序框架

### 超参数搜索

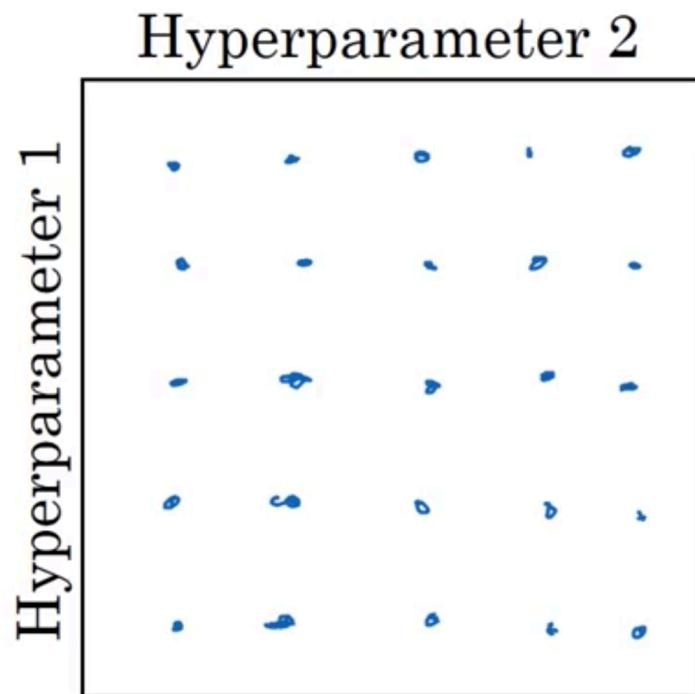
调整超参数对神经网络的优化起到关键性的作用，目前接触过的超参数包括：

1.  $\alpha$
2. unit number
3. layer number
4. momentum or RMSprop's  $\beta$
5. Adam's  $\beta_1$ ,  $\beta_2$ ,  $\epsilon$
6. learning rate decay parameter

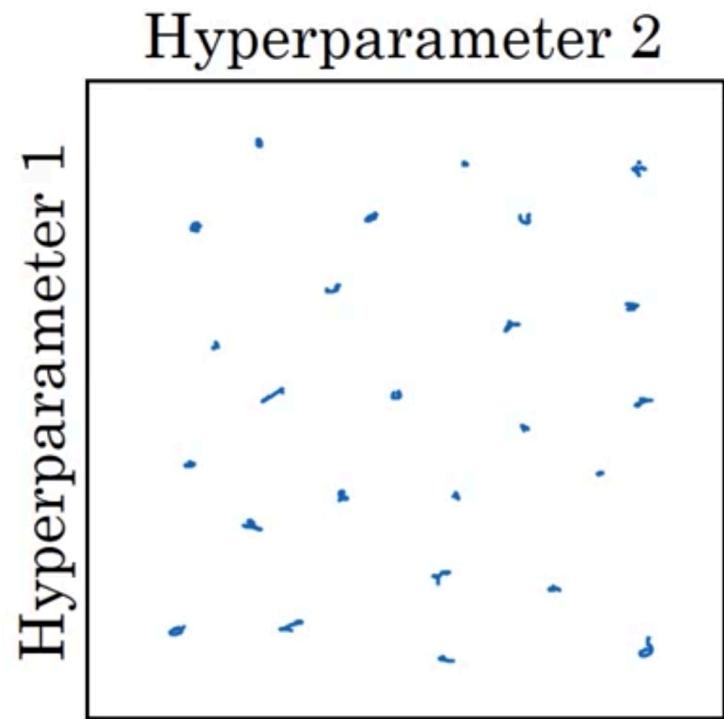
标注红色的为教程中认为很重要的参数，标注橙色或紫色的次之，黑色的为几乎无需调整，使用官方默认设置的参数。我个人认为 batch-size 也可以作为一个超参数。

接下来说说调整超参数的方法。

机器学习时期使用网格调整参数的值，例如对于两个参数  $\alpha$  和  $\epsilon$  构成的如下图的 5\*5 二维网格：

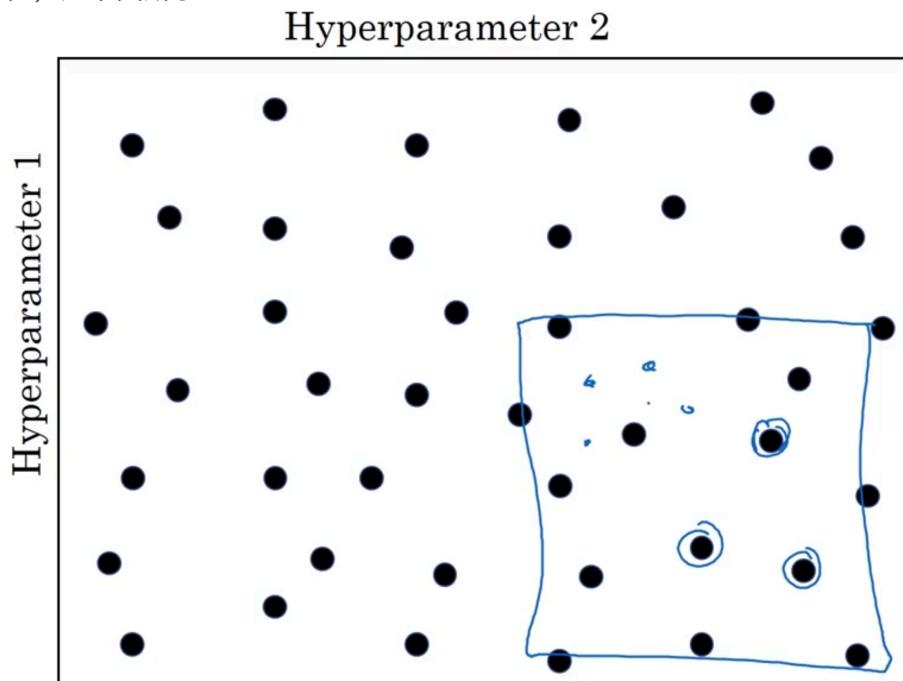


例如图片左侧的 Hyperparameter 1 为  $\alpha$ ，上侧为 Hyperparameter 2 为  $\epsilon$ 。由于对  $\epsilon$  的改变不太会影响到最终的结果，因此，这里实际上是对  $\alpha$  参数的一个选取，但是也仅能尝试 5 种不同类型的  $\alpha$  值。因此，我们使用随机的值，来挑选合适的  $\alpha$ ，如下图：



这样的话，随机取多少个值，就会对  $\alpha$  取多少个值，这样能更快找出合适的  $\alpha$  值。当然这里只是举例，具体需要调整那些超参数，以及维度是多少，需要根据实际情况决定。

关于上述那个随机取值的例子，如果取到的某几个点比较合适，可以进行缩放，继续找值，直到确定比较合适的值，如下图所示：

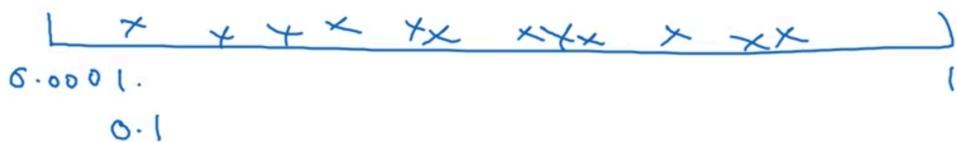


下一节内容，我们会详细讲讲如何选取超参数。

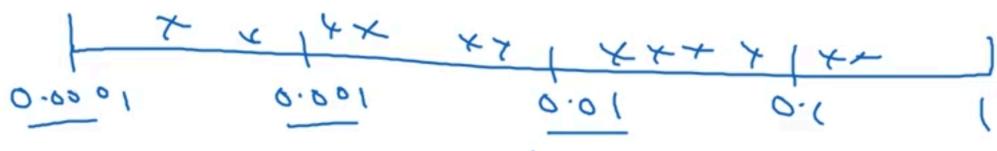
超参数搜索范围标尺的选取

上一节内容我们明确了超参数选取的策略——使用随机选取。但是我们还没有明确如何进行随机选取，尤其是随机选取的范围是什么。本节我们来探讨一下这个内容。

对于随机选取，一般可能会想到使用均匀的标尺进行随机选取。例如在最小值为 0.0001，最大值为 1.0 时，均匀刻度随机选取的情况如下：



这里，不论你的刻度取得多细，随机值取到 0.1 到 1.0 的范围内的概率都会为 90%，而 0.0001 到 0.1 之间的范围为 10%，这其实并不均匀。因此，我们使用非线性的标尺，如下图所示：

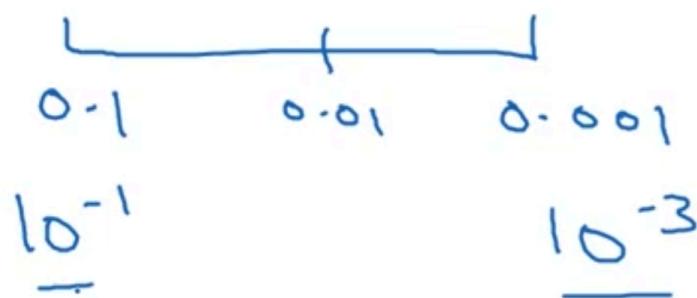


这样能够比较均匀的取到各个区间的值。在 python 中，实现上述思路的方法如下：

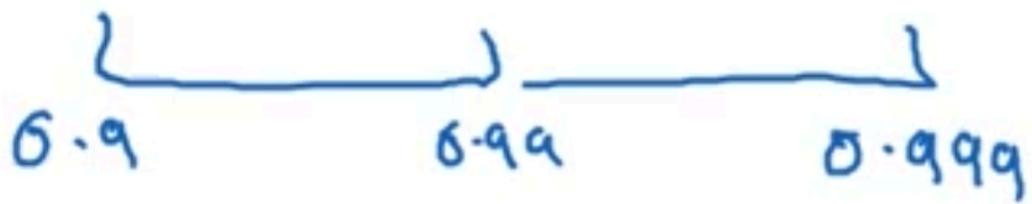
```
1 | r = -4 * np.random.rand() # (1e-4, 1)
2 | parameter = 10 ** r
```

由于 `np.random.rand()` 会返回一个 [0, 1]，所以 r 的范围是在 (-4, 0]，而 parameter 的范围是在  $(10^{-4}, 10^0]$ ，这个方法可以用在  $\alpha$  (学习率) 身上，对于神经网络层数，则直接使用整数随机即可。但是对于 Adam 算法中的  $\beta$  值，这个方法不一定奏效。

因为之前的内容有讲到， $\beta$  值的取值范围在 0.9 到 0.999 之间，因此我们要做的是非线性分割  $1 - \beta$  的范围 [0.001, 1]。按照之前的思路，可以按照下图划分：



在这里其实就可以从另一个角度来理解为什么非线性划分更合适。我们知道  $\beta$  在修正偏差中的公式为  $1/(1 - \beta^2)$ ，对于范围 [0.900, 0.905] 来说，该变动不大，但是对于 [0.995, 0.999] 来说，变化是巨大的，因此如果不按照下图划分：



那么最终的结果会很糟糕。该随机取值的代码实现如下：

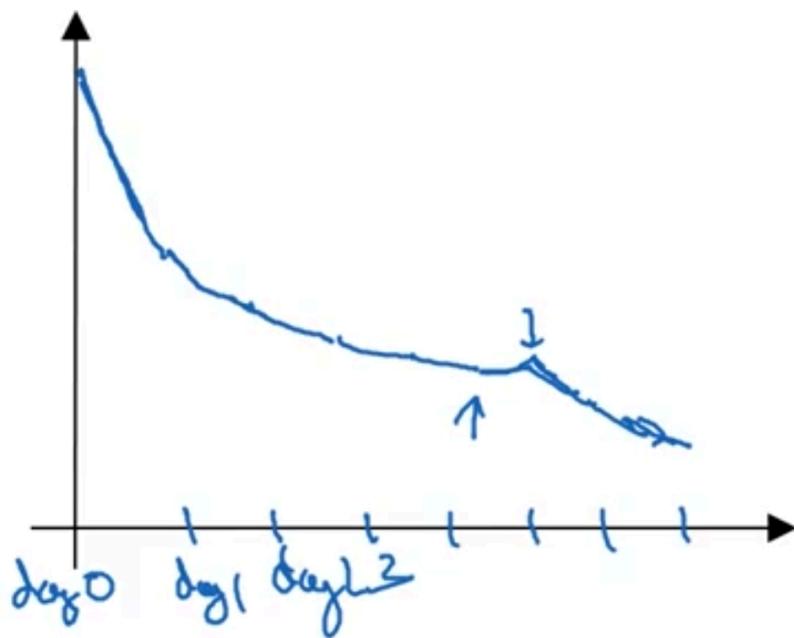
```

1 | r = -3 * np.random.rand()
2 | parameter = 10 ** r
3 | beta = 1 - parameter

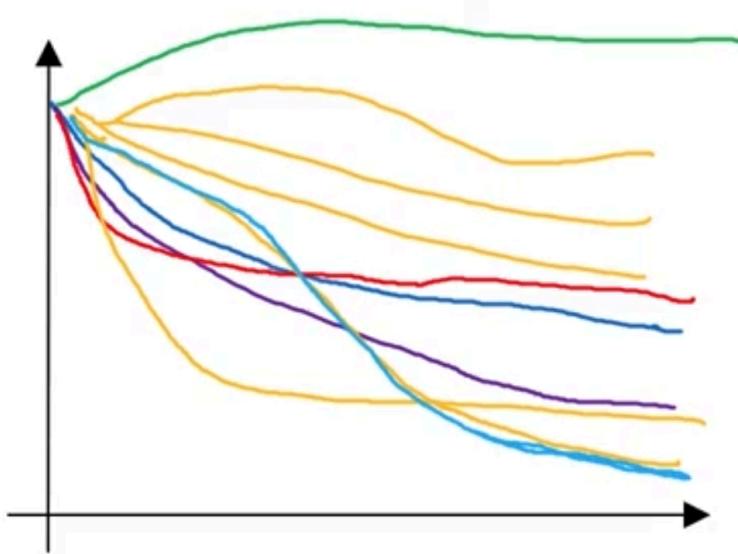
```

### 关于超参数搜索的一些技巧

有些领域的超参数搜索技巧是可以通用的，因此阅读一些其他领域的论文有可能会获得一些灵感。另外，教程中提出，一般搜索超参数的方法大致可以分成两类。一种是 baby sitting 类型的，适用于 CPU/GPU 资源比较匮乏的情况。该方法针对数量较少的模型，一种或两种，不定期地调整超参数，并观察超参数的表现。例如第一天表现的不错，那么尝试增大学习率看看，第二天发现仍旧表现不错，那么为模型添加上 momentum 算法试试，第四天发现情况有点不太好，那么回到第三天的状态，调整算法，再次尝试。上述过程图例如下：



第二种方法是同时训练多个模型，适用于 CPU/GPU 资源充足的情况下，适用多个不同的超参数，训练多个模型，找出表现最好的那个模型。图例如下：



## Batch 归一化

还记得我们在之前提到过的对输入特征归一化的内容吗？我们先计算特征的平均值，然后计算特征的标准差，使输入特征的方差为 1，平均值为 0，使得输入特征均匀分布，提高梯度下降的效率。但实际上在神经网络中，除了输入层之外，在隐层还有激活值  $a$ ，那么对这些激活值进行归一化能不能提升梯度下降的效率？这个想法实际上就是 Batch 归一化。深度学习领域中，对  $a$  归一化和对  $z$  是有争议的，作者在这里默认是对  $z$ 。具体步骤如下所示：

对于  $l$  层的  $Z$ ,  $z^{[l](1)}, z^{[l](2)}, z^{[l](3)}, \dots, z^{[l](i)}$  有：

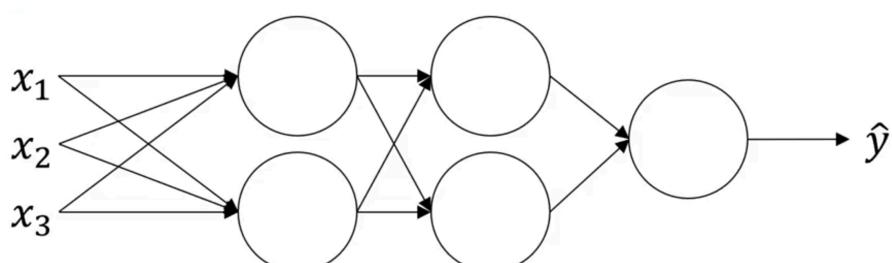
$$\begin{aligned}\mu &= \frac{1}{m} \sum_i z^{[l](i)} \\ \sigma^2 &= \frac{1}{m} \sum_i (z^{[l](i)} - \mu)^2 \\ z_{norm}^{[l]} &= \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{[l]} &= \gamma * z_{norm}^{[l]} + \beta\end{aligned}$$

这里的  $\gamma$  和  $\beta$  实际上是一种学习率，用于调整  $\tilde{z}$  的平均值，我的理解是，由于  $z_{norm}^{[l]}$  分布在 0 点附近，而有时候你不想让这些值处于这种情况（例如激活函数是 sigmoid 时，你不想让函数集中在类线性函数那一段上），就可以使用  $\gamma$  和  $\beta$  进行调整。

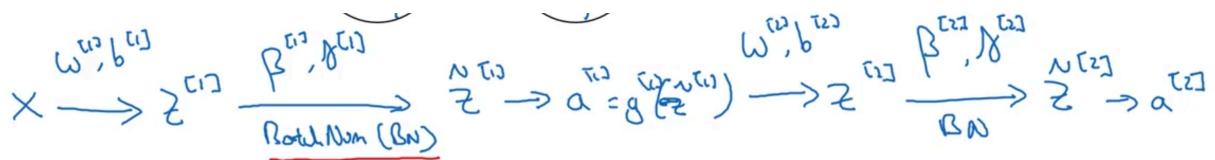
## 对神经网络应用 Batch 归一化

之前提到过对神经网络的一层进行归一化的过程，现在来讲讲对神经网络应用 Batch 归一化的具体步骤。

对于下图这个神经网络：



在应用了 Batch 正则化后，一次神经网络的正向传播过程为，对于输入集  $X$ ，计算第一层的  $z^{[1]}$ ，然后应用 Batch 正则化，通过  $\gamma^{[1]}$  和  $\beta^{[1]}$  计算  $z^{[1]}$ ，再计算激活值  $a^{[1]}$ 。接着通过  $w^{[2]}$  和  $b^{[2]}$  计算  $z^{[2]}$ ，再计算  $z^{[2]}$  以此类推。



当进行反向传播时，作为与  $w^{[l]}$  和  $b^{[l]}$  同等作用的参数  $\gamma^{[l]}$  和  $\beta^{[l]}$ ，也要参与训练，例如，在梯度下降过程中需要计算  $d\gamma^{[l]}$  和  $d\beta^{[l]}$ ，并更新  $\gamma^{[l]}$  和  $\beta^{[l]}$  ( $\gamma^{[l]} = \gamma^{[l]} - \alpha * d\gamma^{[l]}$ ,  $\beta^{[l]} = \beta^{[l]} - \alpha * d\beta^{[l]}$ )，当然也可以使用其他优化算法，例如 Adam, momentum 或者 RMSprop 等等，计算步骤也不发生变化。当然，要注意的是，Batch 归一化中的  $\beta$  与上述三种优化算法中的  $\beta$  不一样。除此之外还有一点需要注意，教程中认为，对  $z^{[l]}$  归一化时， $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$  中的  $b^{[l]}$  其实是起不到作用的，因为再计算均值之后，该值会因为计算均值，减去均值而被抵消，所以在计算 Batch 算法时，可以去掉这个参数，或者可以说是用  $\beta^{[l]}$  来代替（个人认为初始化中，这里的  $b$  确实是一样的，计算均值后会被归零，但在进行一次优化算法更新之后， $b$  向量中的每一个值几乎不一样，我个人认为这里的抵消只能抵消一部分，或者说是  $b$  的作用比较小，编程练习直接把反向传播交给 Tensorflow 了）。因此计算 Batch 归一化时，可以去掉参数  $b$ ，以及更新中的  $db$ 。

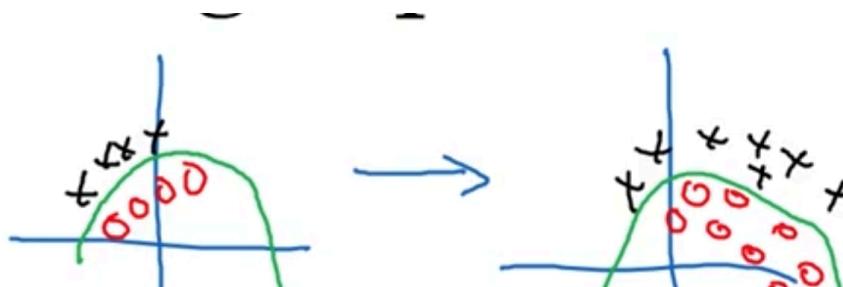
另外，对于参数的维度，维度如下：

$$\begin{aligned}\gamma^{[l]} &: (n^{[l]}, 1) \\ \beta^{[l]} &: (n^{[l]}, 1)\end{aligned}$$

下一节我们来讲讲 Batch 归一化起作用的原因。

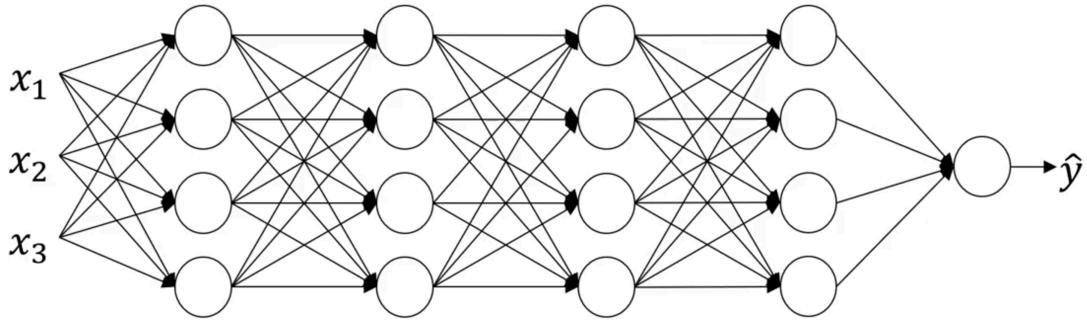
### Batch 的作用及其原因

对于一个神经网络，例如应用在喵脸识别的神经网络，如果你的训练集全部都是黑猫，用训练好的模型去预测除黑色之外颜色的猫，效果肯定是不好的。就如下图所示：

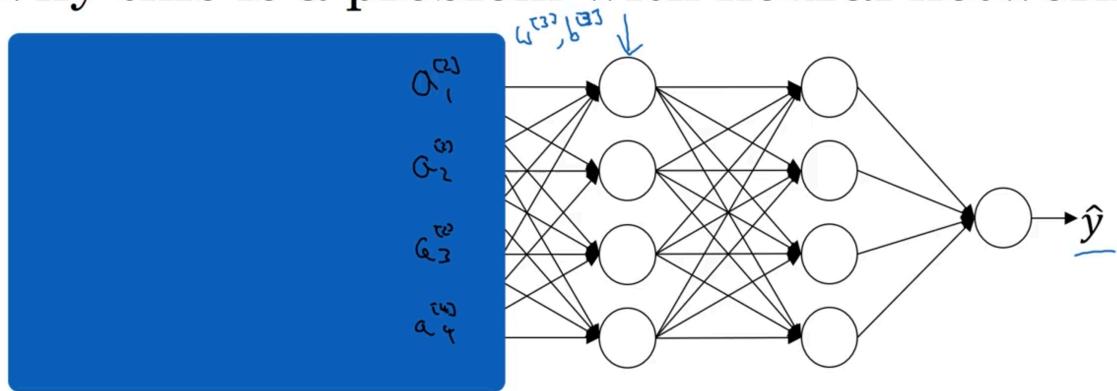


如果想用左侧的函数去预测右侧的数据集，显然效果是不好的。但是这种改变数据分布的思想被称作：covariate shift。这里我们直观的来体会一下 covariate shift 以及 Batch 归一化的作用及其原因。

有下图神经网络



我们从第三层看起，假如遮住前面的输入，将第二层的输出结果定义为  $a^{[2]}$  作为第三层的输入，如下图所示



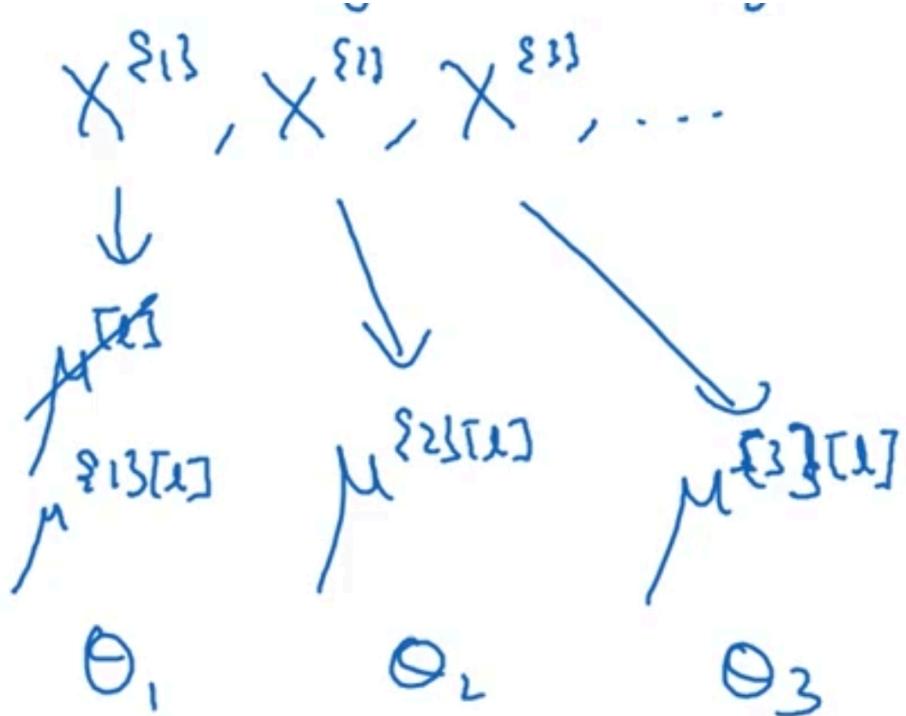
则第三层和第四层以及之后层的作用就是建立一种映射关系，使得预测值更接近真实值。但是由于  $a^{[2]}$  的值受到地一、二层参数的影响，一、二层的参数改变， $a^{[2]}$  同样会改变，对于第三层来说，输入的数据不断地在改变，这就存在 Covariate shift 问题。Batch 归一化的作用就是限制输入数据分布的变化，也就是降低 Covariate shift。之前我们讨论过对输入数据归一化的作用，它使得这些数据的分布比较集中，Batch 归一化所做的事就是如此。它使得输入的激活值变得集中和稳定，降低前层参数变化带来的影响，同时也使得各层之间变得相对独立，这样有助于加速整个神经网络的学习，由于前层数据的方差和平均值不会差太多，使得后层网络的学习变得更加容易。

Batch 归一化的另一个效果是，使得网络被轻微的正则化。之前的内容提到过，正则化的作用是防止神经网络出现过拟合的问题，换句话说，就是使得神经网络变得泛化。由于 Batch 归一化一般作用在 mini batch 算法中，而 mini batch 在进行训练时，由于每一组数据都不完全相同，因此每一层都会计算出不同的平均值和方差，这会使得神经网络具有噪声，一定程度上提高了神经网络的泛化程度。这与 dropout 算法带来的作用一样。因此，混合 dropout 和 Batch 归一化两种算法，能够带来较强的正则化作用。除此之外，mini batch 的 size 也会影响到正则化作用的大小。

### 测试时期的 Batch 归一化

前面提到过，Batch 归一化一般在 mini-Batch 算法的基础下使用，由于测试时期并不一定有 mini-Batch，有可能只有一个数据，并起这个时候针对该数据的均值和方差几乎没有意义，我们应该如何实施 mini-Batch。

通过之前的公式，我们可以计算出每个 mini-Batch 中每个隐藏层的均值和方差，如下图所示：



我们在之前用天气举例子的时候，讲过可以使用指数加权均值的方式预测未来的天气是多少度，这里我们用指数加权平均，通过记录每个 mini-Batch 中每个隐藏层的均值和方差，来预测输入图像的均值和方差。有了均值和方差，就可以计算  $z_{norm}$  以及  $\tilde{z}$ 。该方法可参阅该笔记 [优化深度神经网络 嘿嘿帕斯——CSDN](#)，这里的计算过程教程直接通过之后要讲的 Tensorflow 实现了，因此这里只需要了解下思路就好。

### softmax 回归

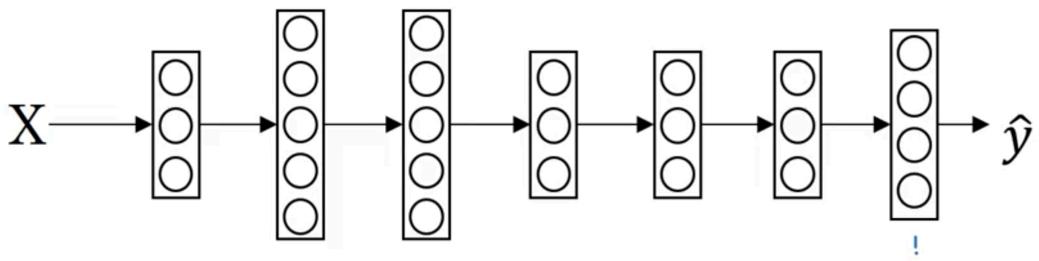
到目前为止，我们所接触的分类器都是在做二元分类，现实中有很多需要多元分类的情况，此时可以使用 softmax 算法。

softmax 算法与二元分类的不同点在于预测值的维度。我们从一个例子来看。

下列图片中，我们将猫，狗，小鸡分别标记为 1, 2, 3，将不属于它们中的任何一类动物/事物标记为 0。



因此，我们的类别数量为 4，那么输入一组图片，输出的预测值应该是这四个维度的预测概率。假如我们针对该问题构建了一个神经网络，如下图：

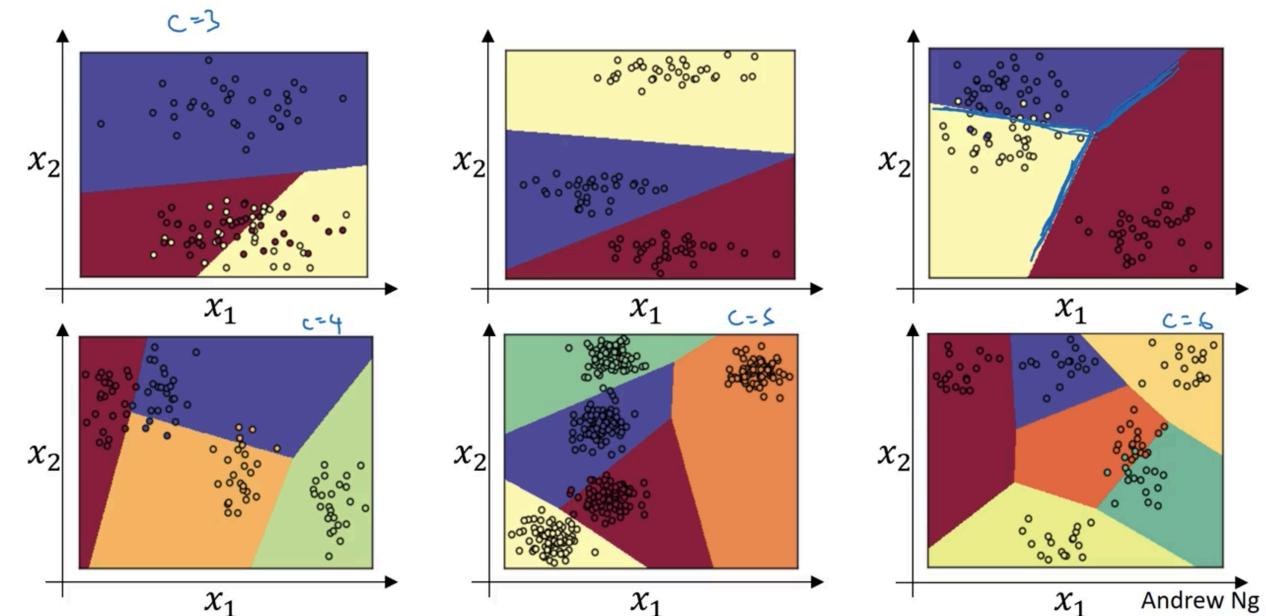


这里的  $\hat{y}$  和  $z^{[L]}$  的维度为  $(4, 1)$ ，不过要注意，softmax 算法在最后一层的激活函数与之前不一样，计算激活值  $a^{[L]}$  的过程如下：

$$t = e^{z^{[L]}}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^4 t_i}$$

一个 softmax 逻辑回归，或者说是 softmax 分类器（没有隐藏层的神经网络）的表现如下图：



关于使用 softmax 作为多元分类问题模型的原因，可以参考这些博客和答案：[每日一问之多元分类为什么使用 softmax 函数 —— CSDN](#)，[多类分类下为什么用softmax而不是用其他归一化方法? —— 知乎](#)

### 关于 softmax 的补充

除了前向传播的上述公式之外，我们还需要计算损失函数以及成本函数。损失函数的公式为

$L(\hat{y}, y) = -\sum_{i=1}^C y^{(i)} \log \hat{y}^{(i)}$ ，但是因为一个图片只属于一个分类，因此除了该分类之外，其他的值均为 0，因此公式变为  $L(\hat{y}, y) = -y^{(i)} \log \hat{y}^{(i)}$ 。例如之前的例子中，某个猫图的标签  $y$  为  $\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$  则，

除了  $i=2$  之外的所有值均为 0。另外，如果要使得损失函数最小，则  $\hat{y}$  应该最大，因此也达到了训练模型的目的。成本函数与之前的公式一致，也是对所有样本的损失函数求和取平均。

前向传播之后还需要进行反向传播，对于反向传播这里只推导一个公式，就是  $\partial L / \partial z^{(j)}$ ，这里的 j 表示第 l 层隐层的第 j 个数据，这里省略 l 层，方便表示。过程如下：

$$\begin{aligned}\frac{\partial L}{\partial z^{(j)}} &= - \sum_{i=1}^C \frac{\partial [y^{(i)} \log(\hat{y}^{(i)})]}{\partial z^{(j)}} \\ &= - \sum_{i=1}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}}\end{aligned}\quad (1)$$

为了方便推导，我们对  $\partial \hat{y}^{(i)} / \partial z^{(j)}$  进行单独讨论，过程如下：

$$\frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} = \frac{\partial \left( \frac{e^{z^{(i)}}}{\sum_{k=1}^C e^{z^{(k)}}} \right)}{\partial z^{(j)}} \quad (2)$$

方便起见，我们令  $\sum = \sum_{k=1}^C e^{z^{(k)}}$ ，并且分情况讨论：

1. 当  $i = j$  时，(2) 式变为

$$\begin{aligned}\frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} &= \frac{\partial \left( \frac{e^{z^{(i)}}}{\sum} \right)}{\partial z^{(j)}} \\ &= \frac{e^{z^{(i)}} \sum - e^{z^{(i)}} e^{z^{(j)}}}{\sum^2} \\ &= \frac{e^{z^{(i)}} \sum - e^{z^{(j)}}}{\sum \sum} \\ &= \hat{y}^{(i)} (1 - \hat{y}^{(j)})\end{aligned}\quad (3)$$

2. 当  $i \neq j$  时，(2) 式变为

$$\begin{aligned}\frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} &= \frac{\partial \left( \frac{e^{z^{(i)}}}{\sum} \right)}{\partial z^{(j)}} \\ &= \frac{0 - e^{z^{(i)}} e^{z^{(j)}}}{\sum^2} \\ &= -\hat{y}^{(i)} \hat{y}^{(j)}\end{aligned}\quad (4)$$

接下来，我们需要整合两个情况到公式 (1) 中

$$\begin{aligned}\frac{\partial L}{\partial z^{(j)}} &= - \sum_{i=1}^C \frac{\partial [y^{(i)} \log(\hat{y}^{(i)})]}{\partial z^{(j)}} \\ &= - \sum_{i=1}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} \\ &= -y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(i)}} - \sum_{i \neq j}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}}\end{aligned}\quad (5)$$

公式 (5) 是将前一步求和分成  $i = j$  和  $i \neq j$  这两种情况，左侧偏导数可以应用公式 (3) 和 (4)，带入公式到 (5) 中结果如下：

$$\begin{aligned}
\frac{\partial L}{\partial z^{(j)}} &= - \sum_{i=1}^C \frac{\partial [y^{(i)} \log(\hat{y}^{(i)})]}{\partial z^{(j)}} \\
&= - \sum_{i=1}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} \\
&= -y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(i)}} - \sum_{i \neq j}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{(j)}} \\
&= -y^{(i)} \frac{1}{\hat{y}^{(i)}} \hat{y}^{(i)} (1 - \hat{y}^{(j)}) - \sum_{i \neq j}^C y^{(i)} \frac{1}{\hat{y}^{(i)}} (-\hat{y}^{(i)} \hat{y}^{(j)}) \\
&= -y^{(i)} + y^{(i)} \hat{y}^{(j)} + \sum_{i \neq j}^C y^{(i)} \hat{y}^{(j)} \\
&= -y^{(i)} + \hat{y}^{(j)} \sum_{i=1}^C y^{(i)} \\
&= \hat{y}^{(j)} - y^{(i)} \\
&= \hat{y}^{(j)} - y^{(j)}
\end{aligned}$$

最后一步能将  $i$  换成  $j$  是因为被提出求和符号的那一项的  $i = j$ 。公式推导参考的是这篇文章：[softmax 反向传播 —— CSDN](#)。

我们在编程练习中则只需要记住前向传播步骤以及对  $dz$  求导的结果，我们在完成编程练习的时候会用到深度学习框架，它会根据前向传播的步骤直接推倒出反向传播，下一节，我们来学习深度学习的框架。

## 深度学习框架

教程中列举到的深度学习框架有 Caffe/Caffe2、CNTK、DL4J、Keras、Lasagne、mxnet、PaddlePaddle、TensorFlow、Theano 和 Torch。教程也给出了选择框架的标准，它取决于你想要制作的模型用途，编程语言以及框架是否开源。

## tensorflow 的使用

对于  $\text{cost} = w^2 - 10w + 25$  的情况，使用梯度下降算法之后可得  $\text{cost}$  取最小时， $w = 5$ ，如何用 tensorflow 来实现这个呢？直接上代码

```

1 import numpy as np
2 import tensorflow.compat.v1 as tf # if your tensorflow version is higher
   than version 1.15.0
3 # import tensorflow as tf # if your tensorflow version is less than
   version 2.0
4
5 tf.disable_v2_behavior() # # if your tensorflow version is higher than
   version 1.15.0
6 coefficients = np.array([[1.], [-20.], [100.]])
7 w = tf.Variable(0.0, dtype=tf.float32)
8 x = tf.placeholder(tf.float32, [3, 1])

```

```

9 # cost = tf.add(tf.add(w**2, tf.multiply(-10.0, w)), 25.0)
10 # cost = w**2 - 10*w + 25
11 cost = x[0]*w**2+x[1]*w+x[2]
12 train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
13
14 init = tf.global_variables_initializer()
15 session = tf.Session()
16 session.run(init)
17 print(session.run(cost, feed_dict={x:coefficients}))

```

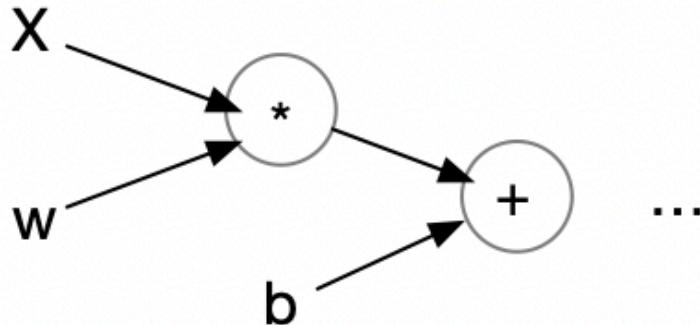
tensorflow 与 python 等编程语言不同，所有被设置的变量在运行时，如果不执行 `init = tf.global_variables_initializer()`，这些变量是不会被初始化的，如果不执行 `session.run(init)`，则初始化不会被执行。可以这样来理解 tensorflow 的运行流程：

定义变量 --> 定义初始化 --> 用 session 执行初始化 --> 用 session 执行其他操作

从上面的流程就可以看出只用使用 session 才能够使参数初始化，执行前向传播，反向传播，或者梯度下降，就如同编译器上的运行按钮。这里简要说一下代码 `x = tf.placeholder(tf.float32, [3, 1])` 为设置占位符，通过 `session.run(train, feed_dict={x:coefficients})` 给占位符填充数据。`cost = x[0]*l**2+x[1]*l+x[2]` 用于设置损失函数 `train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)` 用于定义梯度下降的类型（Batch 梯度下降），学习率（0.01）以及目标（最小化）。`l = tf.Variable(0.0, dtype=tf.float32)` 则是 tensorflow 中定义变量的方式。

Tensorflow 执行运算时，使用的是下面这种计算图的方式：

$$w^*x + b$$



这种计算图与我们之前看到的不太一致，详细内容请参阅官方文档。

```

1 session = tf.Session()
2 session.run(init)

```

上段代码也可以写成：

```
1 | with tf.Session() as session:  
2 |     session.run(init)
```

第二种写法更有利于检错。

## 课后编程练习 1

本编程练习将使用 Tensorflow 搭建神经网络。首先先引入环境

```
1 | import math  
2 | import numpy as np  
3 | import h5py  
4 | import matplotlib.pyplot as plt  
5 | import tensorflow as tf2 # 为了使用 initializers.GlorotUniform 函数  
6 | import tensorflow.compat.v1 as tf  
7 | from tensorflow.python.framework import ops  
8 | from tf_utils import load_dataset, random_mini_batches,  
|   convert_to_one_hot, predict  
9 |  
10 | %matplotlib inline  
11 | np.random.seed(1)
```

与以往一样，我将教程中的源代码文件放在了 `Week3_编程练习/functions` 下，数据则放在了 `Week3_编程练习/datasets` 下。需要注意的是，教程中使用的 Tensorflow 版本至少低于 v2.0，我这里的 Tensorflow 环境为 v2.1，因此需要使用 `import tensorflow.compat.v1 as tf` 来支持教程中的代码，另外还需要 `tf.disable_v2_behavior()` 阻止 v2.0+ 的行为。接下来我们用损失函数  $loss = \mathcal{L}(\hat{y}, y) = (\hat{y}^{(i)} - y^{(i)})^2$  做测试。代码如下：

```
1 | tf.disable_v2_behavior()  
2 | y_hat = tf.constant(36, name='y_hat')           # Define y_hat constant.  
| Set to 36.  
3 | y = tf.constant(39, name='y')                  # Define y. Set to 39  
4 |  
5 | loss = tf.Variable((y - y_hat)**2, name='loss') # Create a variable for  
| the loss  
6 |  
7 | init = tf.global_variables_initializer()        # When init is run later  
| (session.run(init)),  
8 |                                         # the loss variable will  
| be initialized and ready to be computed  
9 | with tf.Session() as session:                  # Create a session and  
| print the output  
10 |    session.run(init)                         # Initializes the  
| variables  
11 |    print(session.run(loss))                 # Prints the loss
```

运行代码得到结果 9。

总结一下，运行一个 Tensorflow 程序需要做以下几个步骤：

1. 创建 Tensorflow 变量
2. 创建 Tensorflow 操作
3. 创建 Tensorflow 初始化
4. 创建 Tensorflow Session
5. 使用 Session 运行操作。

因此，我们如果构建了损失函数，仅仅只是定义了它，并没有执行它，这个在上节也说过了，我们需要定义 init 和 session 并使用 session 运行 init 和 loss 才能真正运行损失函数。我们来看一个简单的例子：

```
1 | a = tf.constant(2)
2 | b = tf.constant(10)
3 | c = tf.multiply(a,b)
4 | print(c)
```

输出为 `Tensor("Mul:0", shape=(), dtype=int32)`，多运行几次之后，`Mul:0` 会逐次增加。实际上可以看出，你得到的并不是 20，而是如同数据类型一样的说明，并且维度为空，数据类型为 `int32`。接下来我们用 `Session` 来运行上述操作

```
1 | sess = tf.Session()
2 | print(sess.run(c))
```

得到的结果就是 20 了。接下来，我们来讲讲 `placeholders`，它的功能是定义一个占位符，可以在后续通过 `feed_dict = {placeholders : value}` 赋值，示例代码如下：

```
1 | x = tf.placeholder(tf.int64, name = 'x')
2 | print(sess.run(2 * x, feed_dict = {x: 3}))
3 | sess.close()
```

运行结果为 6。

了解完基本的 Tensorflow 用法，我们尝试用它编写一个线性函数。尝试计算  $WX+b$ ， $W$ ， $X$  和  $b$  均满足随机正态分布， $W$ ， $X$  和  $b$  的维度分别为  $(4, 3)$ ， $(3, 1)$  和  $(4, 1)$ 。上述的线性函数的 Tensorflow 实现如下：

```
1 | def linear_function():
2 |     np.random.seed(1)
3 |     ### START CODE HERE ### (4 lines of code)
4 |     X = tf.constant(np.random.randn(3, 1), name='X')
5 |     W = tf.constant(np.random.randn(4, 3), name='W')
6 |     b = tf.constant(np.random.randn(4, 1), name='b')
7 |     Y = tf.matmul(W, X) + b
8 |     ### END CODE HERE ###
9 |     # Create the session using tf.Session() and run it with sess.run(...)
10 |    on the variable you want to calculate
11 |    ### START CODE HERE ###
```

```

11     sess = tf.Session()
12     result = sess.run(Y)
13     ##### END CODE HERE #####
14     # close the session
15     sess.close()
16     return result
17
18 print( "result = " + str(linear_function()))

```

运行结果为：

```

1 result = [[-2.15657382]
2 [ 2.95891446]
3 [-1.08926781]
4 [-0.84538042]]

```

Tensorflow 本身集成了 softmax 和 sigmoid 函数，我们接下来来实现 sigmoid 函数，代码如下：

```

1 def sigmoid(z):
2     ##### START CODE HERE ##### ( approx. 4 lines of code)
3     # Create a placeholder for x. Name it 'x'.
4     x = tf.placeholder(tf.float32, name='x')
5     # compute sigmoid(x)
6     sigmoid = tf.sigmoid(x)
7     # Create a session, and run it. Please use the method 2 explained
8     # above.
9     # You should use a feed_dict to pass z's value to x.
10    with tf.Session() as sess:
11        # Run session and call the output "result"
12        result = sess.run(sigmoid, feed_dict={x:z})
13    ##### END CODE HERE #####
14    return result
15
16 print ("sigmoid(0) = " + str(sigmoid(0)))
17 print ("sigmoid(12) = " + str(sigmoid(12)))

```

`with tf.Session() as sess:` 这个写法等同于 `sess = tf.Session()`，上述代码运行结果如下：

```

1 sigmoid(0) = 0.5
2 sigmoid(12) = 0.9999938

```

接下来我们来计算 Cost function。公式如下

$$J = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log a^{[2](i)} + (1 - y^{(i)}) \log(1 - a^{[2](i)}))$$

使用 Tensorflow 框架，上述公式可以用一行代码来实现，实现如下：

```

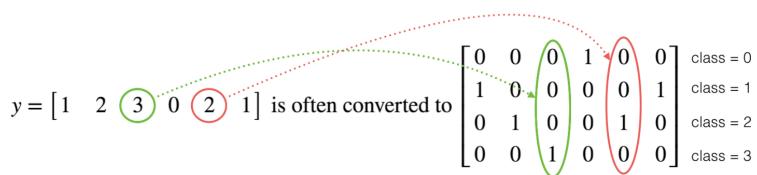
1 def cost(logits, labels):
2     ### START CODE HERE ###
3     # Create the placeholders for "logits" (z) and "labels" (y) (approx. 2
4     # lines)
5     z = tf.placeholder(tf.float32, logits.shape, name='z')
6     y = tf.placeholder(tf.float32, labels.shape, name='y')
7     # Use the loss function (approx. 1 line)
8     cost = tf.nn.sigmoid_cross_entropy_with_logits(logits=z, labels=y)
9     # Create a session (approx. 1 line). See method 1 above.
10    sess = tf.Session()
11    # Run the session (approx. 1 line).
12    cost = sess.run(cost, feed_dict={z:logits, y:labels})
13    # Close the session (approx. 1 line). See method 1 above.
14    sess.close()
15    ### END CODE HERE ###
16
17 logits = sigmoid(np.array([0.2,0.4,0.7,0.9]))
18 cost = cost(logits, np.array([0,0,1,1]))
19 print ("cost = " + str(cost))

```

`tf.nn.sigmoid_cross_entropy_with_logits` 能够直接根据输入的 `z` 和 `y` 计算 cost function，无需计算 sigmoid。计算结果如下：

```
1 | cost = [1.0053872 1.0366409 0.41385433 0.39956614]
```

通常，标签 `y` 中的数据范围为  $[0, C-1]$ ， $C$  为图像类别数量，如果  $C=4$ ，那么 `y` 可能如下图所示：



上面这个转换方式被称为 one hot encoding，中文翻译为独热编码，接着我们来实现这个过程，代码如下所示：

```

1 def one_hot_matrix(labels, C):
2     ### START CODE HERE ###
3     # Create a tf.constant equal to C (depth), name it 'C'. (approx. 1
4     # line)
5     C = tf.constant(C, name='C')
6     # Use tf.one_hot, be careful with the axis (approx. 1 line)
7     one_hot_matrix = tf.one_hot(labels, C, axis=0)
8     # Create the session (approx. 1 line)
9     sess = tf.Session()
10    one_hot = sess.run(one_hot_matrix)

```

```
11 # Close the session (approx. 1 line). See method 1 above.  
12 sess.close()  
13 ### END CODE HERE ###  
14 return one_hot  
15  
16 labels = np.array([1,2,3,0,2,1])  
17 one_hot = one_hot_matrix(labels, C = 4)  
18 print ("one_hot = " + str(one_hot))
```

运行上述代码可得：

```
1 one_hot = [[0. 0. 0. 1. 0. 0.]  
2 [1. 0. 0. 0. 0. 1.]  
3 [0. 1. 0. 0. 1. 0.]  
4 [0. 0. 1. 0. 0. 0.]]
```

注：教程中给出的结果是这个，该结果可通过设置 tf.one\_hot() 函数中的 axis 改变维度

接下来我们实现全 1 初始化，和全 0 初始化的方法，如下段代码所示

```
1 def ones(shape):  
2     ### START CODE HERE ###  
3     # Create "ones" tensor using tf.ones(...). (approx. 1 line)  
4     ones = tf.ones(shape)  
5     # Create the session (approx. 1 line)  
6     sess = tf.Session()  
7     # Run the session to compute 'ones' (approx. 1 line)  
8     ones = sess.run(ones)  
9     # Close the session (approx. 1 line). See method 1 above.  
10    sess.close()  
11    ### END CODE HERE ###  
12    return ones  
13  
14 print ("ones = " + str(ones([3])))
```

运行结果如下：

```
1 | ones = [1. 1. 1.]
```

接下来我们要使用 Tensorflow 构建一个神经网络，用于识别手语，来帮助语言障碍者和不懂手语的人进行交流。教程给了下图作为示例，个人感觉是做手语的数字识别。

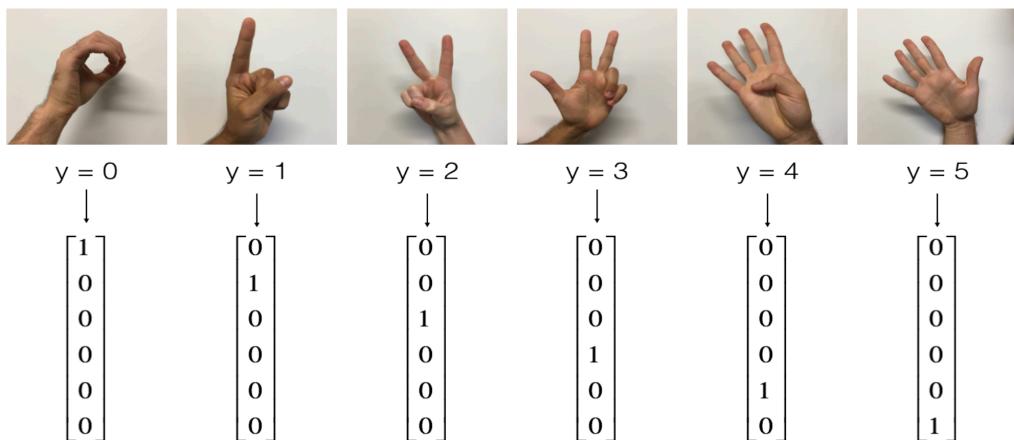


Figure 1: SIGNS dataset

我们先加载数据集，数据集所在的文件夹为 Week3\_编程练习/datasets，代码如下：

```
1 | X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes =
  | load_dataset()
```

可以通过修改下段代码的 index 来查看不同的图像

```
1 | index = 3
2 | plt.imshow(X_train_orig[index])
3 | print ("y = " + str(np.squeeze(Y_train_orig[:, index])))
```

之后，我们通过下段代码构建训练集

```
1 | # Flatten the training and test images
2 | X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
3 | X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T
4 | # Normalize image vectors
5 | X_train = X_train_flatten/255.
6 | X_test = X_test_flatten/255.
7 | # Convert training and test labels to one hot matrices
8 | Y_train = convert_to_one_hot(Y_train_orig, 6)
9 | Y_test = convert_to_one_hot(Y_test_orig, 6)
10 |
11 | print ("number of training examples = " + str(X_train.shape[1]))
12 | print ("number of test examples = " + str(X_test.shape[1]))
13 | print ("X_train shape: " + str(X_train.shape))
14 | print ("Y_train shape: " + str(Y_train.shape))
15 | print ("X_test shape: " + str(X_test.shape))
16 | print ("Y_test shape: " + str(Y_test.shape))
```

运行结果如下：

```
1 number of training examples = 1080
2 number of test examples = 120
3 X_train shape: (12288, 1080)
4 Y_train shape: (6, 1080)
5 X_test shape: (12288, 120)
6 Y_test shape: (6, 120)
```

我们知道，在训练模型的时候，输入的X和Y都在不断的变化，因此，我们需要为X和Y设置一个placeholder，我们构建一个函数来实现这个功能，函数如下：

```
1 def create_placeholders(n_x, n_y):
2     ### START CODE HERE ### (approx. 2 lines)
3     X = tf.placeholder(dtype=tf.float32, shape=[n_x, None], name='X')
4     Y = tf.placeholder(dtype=tf.float32, shape=[n_y, None], name='Y')
5     ### END CODE HERE ###
6     return X, Y
7
8 X, Y = create_placeholders(12288, 6)
9 print ("X = " + str(X))
10 print ("Y = " + str(Y))
```

运行结果如下：

```
1 X = Tensor("X_14:0", shape=(12288, ?), dtype=float32)
2 Y = Tensor("X_15:0", shape=(6, ?), dtype=float32)
```

下一步就是初始化参数，教程中初始化了3层隐藏层的参数W和b，初始化函数用的是tf.contrib.layers.xavier\_initializer(seed=1)，它能够使得每一层的梯度大小差不多相似，具体说明和用法请参阅[官方说明——Github——tf.contrib.layers.xavier\\_initializer](#)。但是，这里对W使用的初始化函数与教程相异，教程中的初始化器为 initializer = `tf.contrib.layers.xavier_initializer(seed = 1)`，然而在Tensorflow v2.0（甚至更早，印象中好像是v1.4）中移除了contrib，集成到核心中去了，同时，至少在v2.0中，没有找到xavier\_initializer()，这个初始化器，因此这里使用initializers.GlorotUniform代替之前的函数。,代码如下：

```
1 def initialize_parameters():
2     """
3         Initializes parameters to build a neural network with tensorflow. The
4         shapes are:
5             W1 : [25, 12288]
6             b1 : [25, 1]
7             W2 : [12, 25]
8             b2 : [12, 1]
9             W3 : [6, 12]
10            b3 : [6, 1]
11
12        Returns:
```

```

12     parameters -- a dictionary of tensors containing W1, b1, W2, b2, W3,
b3
13     """
14
15     tf.set_random_seed(1) # so that your "random"
numbers match ours
16     ### START CODE HERE ### (approx. 6 lines of code)
17     W1 = tf.get_variable('W1', [25, 12288],
initializer=tf.initializers.GlorotUniform(seed=1))
18     b1 = tf.get_variable("b1", [25,1], initializer =
tf.zeros_initializer())
19     W2 = tf.get_variable('W2', [12, 25],
initializer=tf.initializers.GlorotUniform(seed=1))
20     b2 = tf.get_variable("b2", [12,1], initializer =
tf.zeros_initializer())
21     W3 = tf.get_variable('W3', [6, 12],
initializer=tf.initializers.GlorotUniform(seed=1))
22     b3 = tf.get_variable("b3", [6,1], initializer =
tf.zeros_initializer())
23     ### END CODE HERE ###
24
25     parameters = {"W1": W1,
26                 "b1": b1,
27                 "W2": W2,
28                 "b2": b2,
29                 "W3": W3,
30                 "b3": b3}
31
32     return parameters

```

运行结果如下：

```

1 W1 = <tf.Variable 'W1:0' shape=(25, 12288) dtype=float32_ref>
2 b1 = <tf.Variable 'b1:0' shape=(25, 1) dtype=float32_ref>
3 W2 = <tf.Variable 'W2:0' shape=(12, 25) dtype=float32_ref>
4 b2 = <tf.Variable 'b2:0' shape=(12, 1) dtype=float32_ref>

```

接下来编写前向传播的函数：

```

1 def forward_propagation(X, parameters):
2     # Retrieve the parameters from the dictionary "parameters"
3     W1 = parameters['W1']
4     b1 = parameters['b1']
5     W2 = parameters['W2']
6     b2 = parameters['b2']
7     W3 = parameters['W3']
8     b3 = parameters['b3']
9

```

```

10     ### START CODE HERE ### (approx. 5 lines)           # Numpy
11     Equivalents:
12         Z1 = tf.matmul(W1, X) + b1
13         # Z1 = np.dot(W1, X) + b1
14         A1 = tf.nn.relu(Z1)                           # A1
15         = relu(Z1)
16         Z2 = tf.matmul(W2, A1) + b2
17         # Z2 = np.dot(W2, a1) + b2
18         A2 = tf.nn.relu(Z2)                           # A2
19         = relu(Z2)
20         Z3 = tf.matmul(W3, A2) + b3
21         # Z3 = np.dot(W3,Z2) + b3
22     ### END CODE HERE ###
23
24     return Z3
25
26 tf.reset_default_graph()
27
28 with tf.Session() as sess:
29     X, Y = create_placeholders(12288, 6)
30     parameters = initialize_parameters()
31     Z3 = forward_propagation(X, parameters)
32     print("Z3 = " + str(Z3))

```

测试结果如下所示

```
1 | Z3 = Tensor("add_2:0", shape=(6, ?), dtype=float32)
```

可以发现，与之前不同的是，我们的前向传播没有输出任何的 cache，这个在后面会做出解释。下面我们来实现 Cost funciton

```

1 def compute_cost(Z3, Y):
2     # to fit the tensorflow requirement for
3     tf.nn.softmax_cross_entropy_with_logits(...,...)
4     logits = tf.transpose(Z3)
5     labels = tf.transpose(Y)
6
7     ### START CODE HERE ### (1 line of code)
8     cost =
9     tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
10 labels=labels))
11    ### END CODE HERE ###
12    return cost
13
14 tf.reset_default_graph()
15 with tf.Session() as sess:
16     X, Y = create_placeholders(12288, 6)
17     parameters = initialize_parameters()

```

```
15     z3 = forward_propagation(X, parameters)
16     cost = compute_cost(z3, Y)
17     print("cost = " + str(cost))
```

输出结果为：

```
1 | cost = Tensor("Mean:0", shape=(), dtype=float32)
```

最后，由于应用了 Tensorflow 框架后，我们不再需要自己实现反向传播，因此我们可以直接构建模型啦！这就是为什么我们不需要使用 cache。构建模型代码如下：

```
1 def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,
2             num_epochs = 1500, minibatch_size = 32, print_cost = True):
3     """
4         Implements a three-layer tensorflow neural network: LINEAR->RELU-
5         >LINEAR->RELU->LINEAR->SOFTMAX.
6
7     Arguments:
8         X_train -- training set, of shape (input size = 12288, number of
9             training examples = 1080)
10        Y_train -- test set, of shape (output size = 6, number of training
11            examples = 1080)
12        X_test -- training set, of shape (input size = 12288, number of
13            training examples = 120)
14        Y_test -- test set, of shape (output size = 6, number of test
15            examples = 120)
16        learning_rate -- learning rate of the optimization
17        num_epochs -- number of epochs of the optimization loop
18        minibatch_size -- size of a minibatch
19        print_cost -- True to print the cost every 100 epochs
20
21    Returns:
22        parameters -- parameters learnt by the model. They can then be used
23        to predict.
24
25        ops.reset_default_graph()                      # to be able to
26        rerun the model without overwriting tf variables
27        tf.set_random_seed(1)                          # to keep
28        consistent results
29        seed = 3                                    # to keep
30        consistent results
31        (n_x, m) = X_train.shape                   # (n_x: input size,
32        m : number of examples in the train set)
33        n_y = Y_train.shape[0]                     # n_y : output size
34        costs = []                                  # To keep track of
35        the cost
36
```

```

27     # Create Placeholders of shape (n_x, n_y)
28     ### START CODE HERE ### (1 line)
29     X, Y = create_placeholders(n_x, n_y)
30     ### END CODE HERE ###
31
32     # Initialize parameters
33     ### START CODE HERE ### (1 line)
34     parameters = initialize_parameters()
35     ### END CODE HERE ###
36
37     # Forward propagation: Build the forward propagation in the
38     # tensorflow graph
39     ### START CODE HERE ### (1 line)
40     Z3 = forward_propagation(X, parameters)
41     ### END CODE HERE ###
42
43     # Cost function: Add cost function to tensorflow graph
44     ### START CODE HERE ### (1 line)
45     cost = compute_cost(Z3, Y)
46     ### END CODE HERE ###
47
48     # Backpropagation: Define the tensorflow optimizer. Use an
49     # AdamOptimizer.
50     ### START CODE HERE ### (1 line)
51     optimizer =
52         tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
53     ### END CODE HERE ###
54
55     # Initialize all the variables
56     init = tf.global_variables_initializer()
57
58     # Start the session to compute the tensorflow graph
59     with tf.Session() as sess:
60
61         # Run the initialization
62         sess.run(init)
63
64         # Do the training loop
65         for epoch in range(num_epochs):
66
67             epoch_cost = 0.                      # Defines a cost
68             related to an epoch
69             num_minibatches = int(m / minibatch_size) # number of
70             minibatches of size minibatch_size in the train set
71             seed = seed + 1
72             minibatches = random_mini_batches(X_train, Y_train,
73             minibatch_size, seed)
74
75             for minibatch in minibatches:

```

```

70
71         # Select a minibatch
72         (minibatch_X, minibatch_Y) = minibatch
73
74         # IMPORTANT: The line that runs the graph on a minibatch.
75         # Run the session to execute the "optimizer" and the
76         # "cost", the feeddict should contain a minibatch for (X,Y).
77         ### START CODE HERE ### (1 line)
78         _, minibatch_cost = sess.run([optimizer, cost],
79         feed_dict={X:minibatch_X, Y:minibatch_Y})
80             ### END CODE HERE ###
81
82         epoch_cost += minibatch_cost / num_minibatches
83
84         # Print the cost every epoch
85         if print_cost == True and epoch % 100 == 0:
86             print ("Cost after epoch %i: %f" % (epoch, epoch_cost))
87         if print_cost == True and epoch % 5 == 0:
88             costs.append(epoch_cost)
89
90         # plot the cost
91         plt.plot(np.squeeze(costs))
92         plt.ylabel('cost')
93         plt.xlabel('iterations (per tens)')
94         plt.title("Learning rate =" + str(learning_rate))
95         plt.show()
96
97         # lets save the parameters in a variable
98         parameters = sess.run(parameters)
99         print ("Parameters have been trained!")
100
101
102         # Calculate the correct predictions
103         correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))
104
105         # Calculate accuracy on the test set
106         accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
107
108         print ("Train Accuracy:", accuracy.eval({X: X_train, Y:
109         Y_train}))
110         print ("Test Accuracy:", accuracy.eval({X: X_test, Y: Y_test}))
111
112         return parameters
113
114     parameters = model(X_train, Y_train, X_test, Y_test)

```

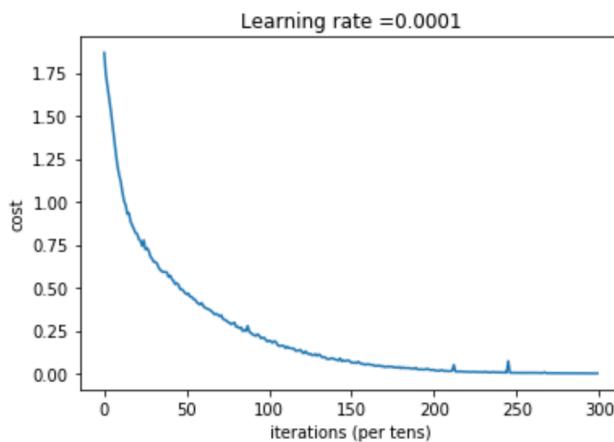
运行结果如下

```

1 | Cost after epoch 0: 1.866523
2 | Cost after epoch 100: 0.812711

```

```
3 Cost after epoch 200: 0.571792
4 Cost after epoch 300: 0.396520
5 Cost after epoch 400: 0.272929
6 Cost after epoch 500: 0.193336
7 Cost after epoch 600: 0.125884
8 Cost after epoch 700: 0.083818
9 Cost after epoch 800: 0.055877
10 Cost after epoch 900: 0.031910
11 Cost after epoch 1000: 0.021902
12 Cost after epoch 1100: 0.011802
13 Cost after epoch 1200: 0.008730
14 Cost after epoch 1300: 0.005802
15 Cost after epoch 1400: 0.003496
```



Parameters have been trained!  
Train Accuracy: 1.0  
Test Accuracy: 0.84166664

这个初始化函数的效果，比教程中给出的要好一点，教程中训练后的结果如下

**Train Accuracy** 0.999074

**Test Accuracy** 0.716667

然后大家可以根据自己拍的图片去测试，我把自己拍的几张测试图片放在 Week3\_编程练习/TestImages 文件夹下了，正确率感人（貌似只有两张图片正确），测试代码如下：

```
1 from PIL import Image
2 from scipy import ndimage
3 import scipy.misc as misc
4 import imageio
5 from skimage.transform import resize
6
7
8 ## START CODE HERE ## (PUT YOUR IMAGE NAME)
9 my_image = "Test9.JPG"
10 ## END CODE HERE ##
11
12 # We preprocess your image to fit your algorithm.
13 fname = "images/" + my_image
14 image = np.array(imageio.imread(fname))
```

```
15 # my_image = misc.imresize(image, size=(64,64)).reshape((1, 64*64*3)).T  
16 my_image = resize(image, output_shape=(64, 64)).reshape((1, 64*64*3)).T  
17 my_image_prediction = predict(my_image, parameters)  
18  
19 plt.imshow(image)  
20 print("Your algorithm predicts: y = " +  
      str(np.squeeze(my_image_prediction)))
```

上述代码适应于最新的各种包，由于教程是 2016 年左右的，因此许多包的函数与以前都大不相同了，不过好在废了一番功夫总算弄成能用的了。至此本周课程也结束了。