计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：Improving Deep Neural Networks: Initialization, Gradient Checking, and Optimization		学号：201900301174
日期：2021-10-12	班级： 智能 19	姓名： 韩旭
Email：hanx@mail.sdu.edu.cn		

实验目的：

Introduction

In this assignment you will master basic neural network adjustment skills and try to improve deep neural networks: Hyperparameter tuning, Regularization and Optimization.

- this time you will be given three subtasks in folder "Improving Deep Neural Networks: Initialization, Gradient Checking, and Optimization".

实验软件和硬件环境：

Termius

Jupyter notebook

RTX3090

Xeon Gold 6226R

实验原理和方法：

- **Setup**
  - **Start IPython**

    you should start the IPython notebook from the homework_2_1 directory, with the jupyter notebook command.
- **Experiments**

  There are ### START CODE HERE/### END CODE HERE tags denoting the start and end of code sections you should fill out. Take care to not delete or modify these tags, or your assignment may not be properly graded.
  - **Q1: Initialization (30 points)**
    The IPython Notebook Initialization.ipynb will walk you through implementing the weight Initialization.
  - **Q2: Gradient Checking (30 points)**
    The IPython Notebook Gradient Checking.ipynb will walk you through implementing the Gradient Checking.
  - **Q3: Optimization (40 points)**
    The IPython Notebook Optimization methods.ipynb will walk you through implementing the optimization algorithms.
  - **See the code file for details.**

实验步骤：（不要求罗列完整源代码）

Initialization

1 Initialization

Welcome to the first assignment of "Improving Deep Neural Networks".

Training your neural network requires specifying an initial value of the weights. A well chosen initialization method will help learning.

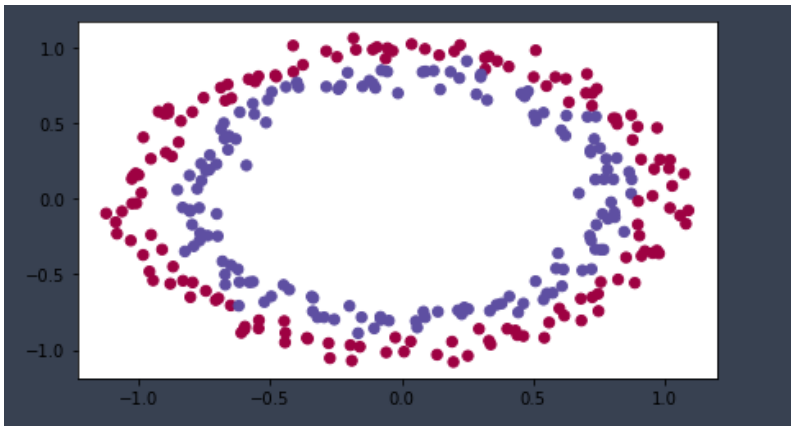
If you completed the previous course of this specialization, you probably followed our instructions for weight initialization, and it has worked out so far. But how do you choose the initialization for a new neural network? In this notebook, you will see how different initializations lead to different results.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

To get started, run the following cell to load the packages and the planar dataset you will try to classify.

首先，我们查看数据分布：



我们发现需要一个非线性的分类器，于是构建了一个神经网络分类器。

初始化网络的参数的时候，我们有三种选择：**Zero initialization**、**Random initialization**、**He initialization**，接下来我们看一下三种初始化方法的效果：

1.2 2 - Zero initialization

There are two types of parameters to initialize in a neural network:

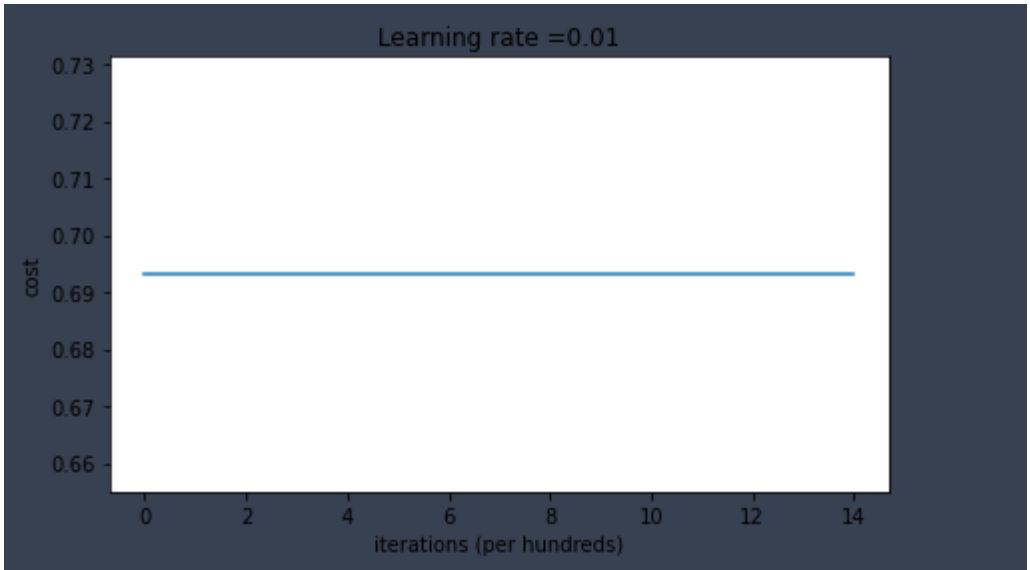
- the weight matrices $(W^{[1]}, W^{[2]}, W^{[3]}, \ldots, W^{[L-1]}, W^{[L]})$
- the bias vectors $(b^{[1]}, b^{[2]}, b^{[3]}, \ldots, b^{[L-1]}, b^{[L]})$

Exercise: Implement the following function to initialize all parameters to zeros. You'll see later that this does not work well since it fails to "break symmetry", but lets try it anyway and see what happens. Use np.zeros((..,..)) with the correct shapes.

代码补充：`initialize_parameters_zeros(layers_dims)`：
其实就是将参数 W 和 b 都初始化为 0，只要注意各自的维度即可。

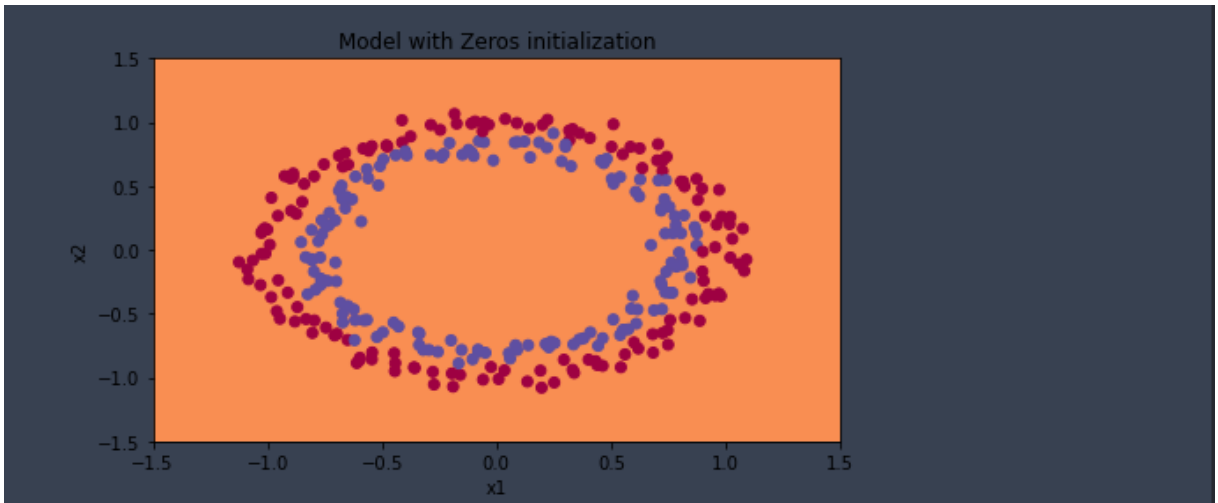
```python
def initialize_parameters_zeros(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    parameters = {}
    L = len(layers_dims)            # number of layers in the network

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.zeros((layers_dims[l],layers_dims[l-1]))
        parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
        ### END CODE HERE ###
    return parameters
```

结果：




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

我们发现网络没有收敛，二分类准确度一直是 0.5，参数一直是 0 没有变，可见分类效果非常不好。



The model is predicting 0 for every example.

In general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and you might as well be training a neural network with $n^{[l]} = 1$ for every layer, and the network is no more powerful than a linear classifier such as logistic regression.

发现网络的参数没有发生改变，把所有的样本归到一类，也就是网络“躺平了”。

1.3 3 - Random initialization

To break symmetry, lets intialize the weights randomly. Following random initialization, each neuron can then proceed to learn a different function of its inputs. In this exercise, you will see what happens if the weights are intialized randomly, but to very large values.

Exercise: Implement the following function to initialize your weights to large random values (scaled by *10) and your biases to zeros. Use `np.random.randn(..,..) * 10` for weights and `np.zeros((.., ..))` for biases. We are using a fixed `np.random.seed(..)` to make sure your "random" weights match ours, so don't worry if running several times your code gives you always the same initial values for the parameters.

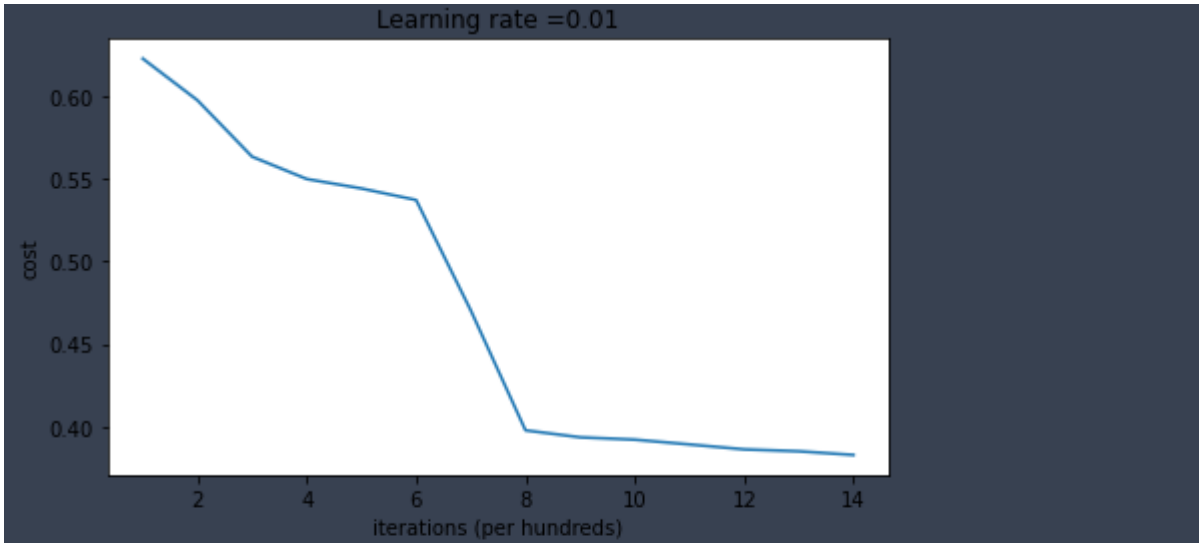
代码补充：

```
initialize_parameters_random(layers_dims)
```

其实就是在上一个零初始化的基础上把 W 改为初始化为[0,10）的符合正态分布的随机数。

```python
def initialize_parameters_random(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    np.random.seed(3)               # This seed makes sure your "random" numbers will be the as ours
    parameters = {}
    L = len(layers_dims)            # integer representing the number of layers

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.random.randn(layers_dims[l],layers_dims[l-1])*10
        parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
        ### END CODE HERE ###

    return parameters
```

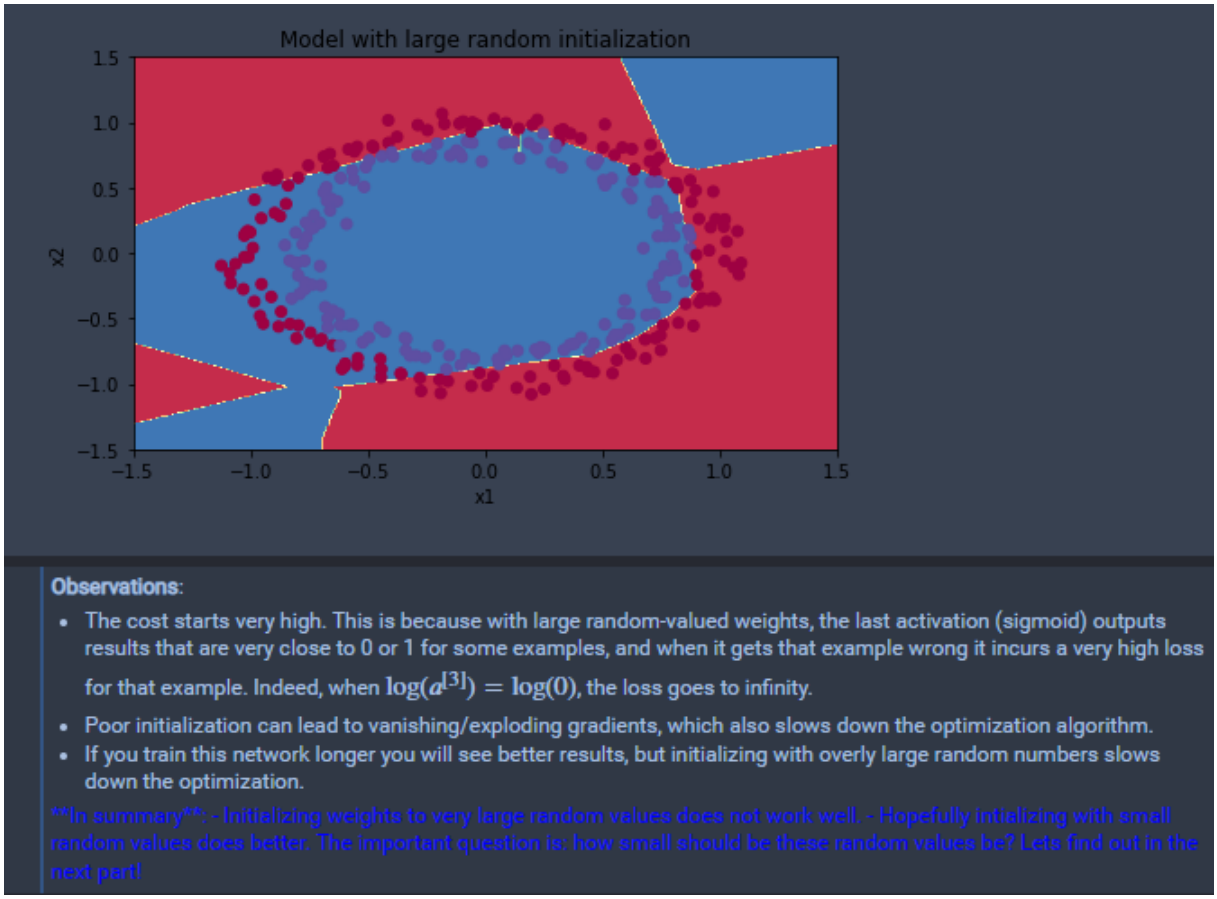
训练过程：




```
On the train set:
Accuracy: 0.83
On the test set:
Accuracy: 0.86
```

我们发现网络可以收敛而且训练集和测试集的准确度都远高于零初始化，都大于 80%。

结果可视化：



Observations:

- The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example. Indeed, when $\log(a^{[3]}) = \log(0)$, the loss goes to infinity.
- Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.
- If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.

In summary: - Initializing weights to very large random values does not work well. - Hopefully intializing with small random values does better. The important question is: how small should be these random values be? Lets find out in the next part!

观察发现，随机初始化得到的分类边界要比零初始化好得多，可以得到大致准确的分类边界。

1.4 4 - He initialization

Finally, try "He Initialization"; this is named for the first author of He et al., 2015. (If you have heard of "Xavier initialization", this is similar except Xavier initialization uses a scaling factor for the weights $W^{[l]}$ of `sqrt(1./layers_dims[l-1])` where He initialization would use `sqrt(2./layers_dims[l-1])`.)

Exercise: Implement the following function to initialize your parameters with He initialization.

Hint: This function is similar to the previous `initialize_parameters_random(...)`. The only difference is that instead of multiplying `np.random.randn(..,..)` by 10, you will multiply it by $\sqrt{\frac{2}{\text{dimension of the previous layer}}}$, which is what He initialization recommends for layers with a ReLU activation.

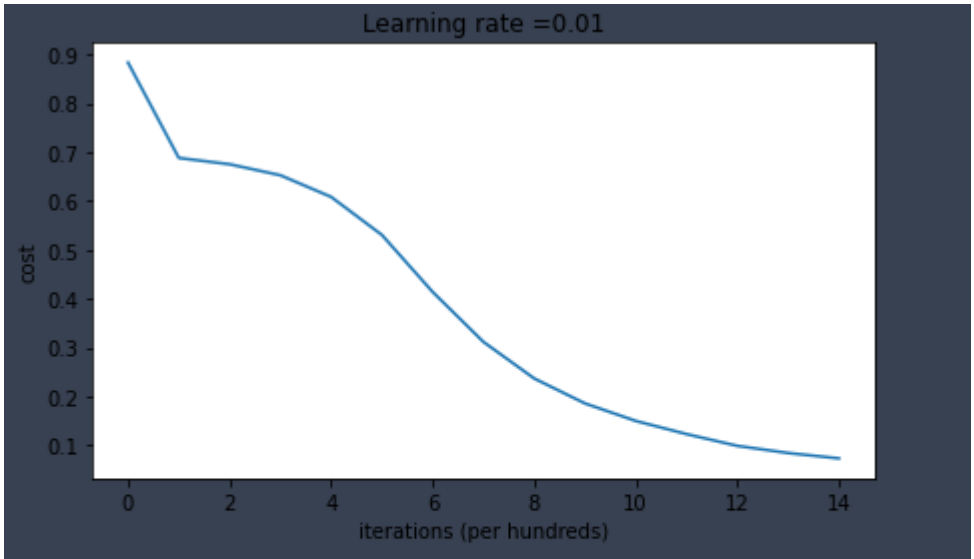
代码补充：

```
initialize_parameters_he(layers_dims)
```

这里其实就是在随机初始化是生成[0,10)随机数的基础上，这里是生成[0,sqrt(2./layers_dims[l-1]))的随机数。

```python
def initialize_parameters_he(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims) - 1 # integer representing the number of layers

    for l in range(1, L + 1):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.random.randn(layers_dims[l],layers_dims[l-1])*np.sqrt(2./layers_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
        ### END CODE HERE ###

    return parameters
```

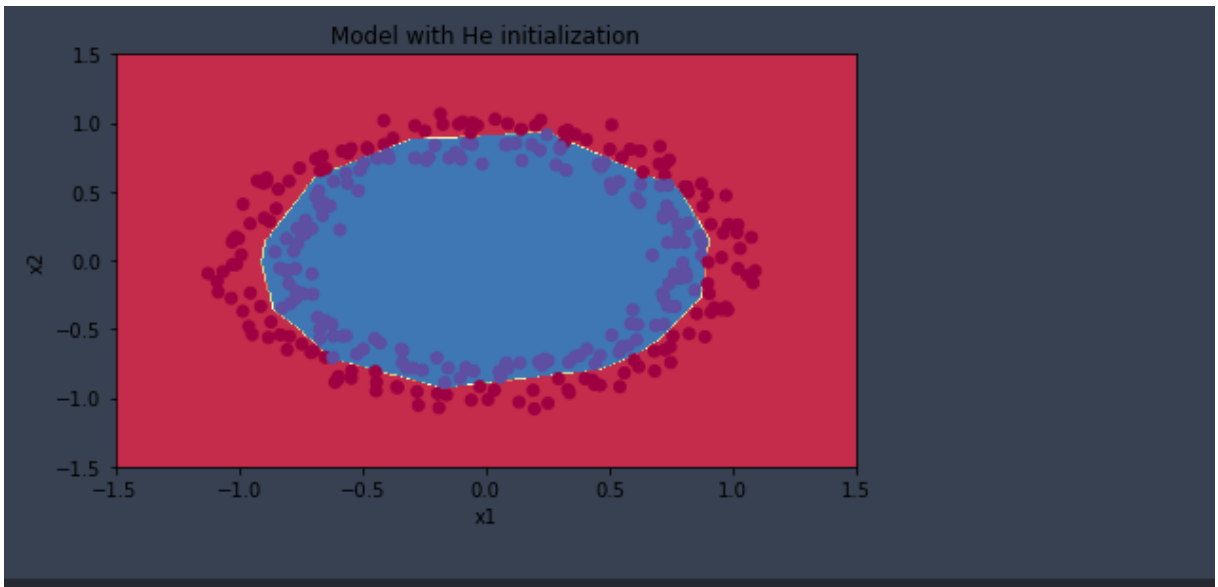
训练过程：




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

观察发现 He 初始化的结果更好，训练准确度能达到 99%，测试集能达到 96%

结果可视化：



Observations:

- The model with He initialization separates the blue and the red dots very well in a small number of iterations.

观察发现 He 初始化得到的分类边界能够很好的把两种样本分离开。

总结：

1.5 5 - Conclusions

You have seen three different types of initializations. For the same number of iterations and same hyperparameters the comparison is:

Model	**Train accuracy**	**Problem/Comment**
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

What you should remember from this notebook: - Different initializations lead to different results - Random initialization is used to break symmetry and make sure different hidden units can learn different things - Don't initialize to values that are too large - He initialization works well for networks with ReLU activations.

Gradient Checking

1 Gradient Checking

Welcome to the final assignment for this week! In this assignment you will learn to implement and use gradient checking.

You are part of a team working to make mobile payments available globally, and are asked to build a deep learning model to detect fraud--whenever someone makes a payment, you want to see if the payment might be fraudulent, such as if the user's account has been taken over by a hacker.

But backpropagation is quite challenging to implement, and sometimes has bugs. Because this is a mission-critical application, your company's CEO wants to be really certain that your implementation of backpropagation is correct. Your CEO says, "Give me a proof that your backpropagation is actually working!" To give this reassurance, you are going to use "gradient checking".

Let's do it!

1.2 2) 1-dimensional gradient checking

Consider a 1D linear function $J(\theta) = \theta x$. The model contains only a single real-valued parameter θ, and takes x as input.

You will implement code to compute $J(.)$ and its derivative $\frac{\partial J}{\partial \theta}$. You will then use gradient checking to make sure your derivative computation for J is correct.

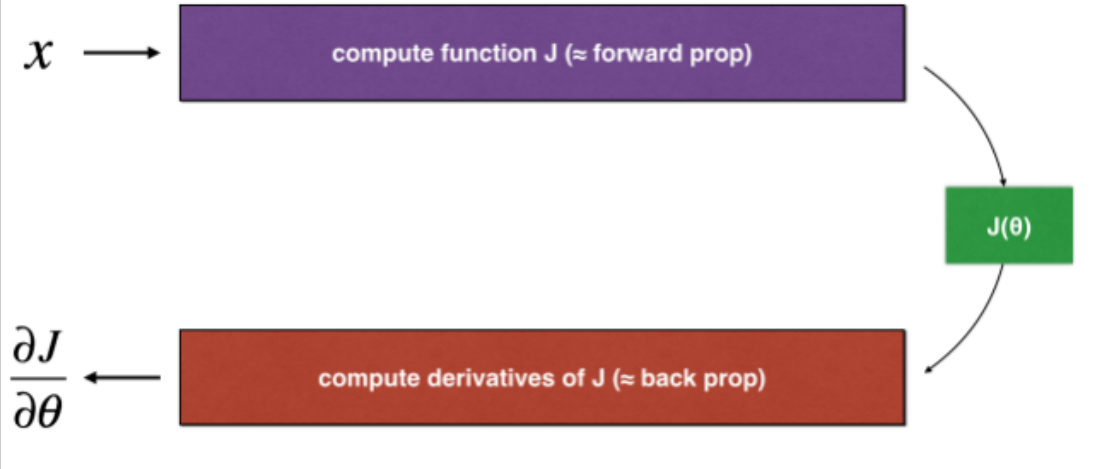


Figure 1: **1D linear model**

补充代码：根据图片补充代码

forward_propagation(x, theta)

Exercise: implement "forward propagation" and "backward propagation" for this simple function. I.e., compute both $J(.)$ ("forward propagation") and its derivative with respect to θ ("backward propagation"), in two separate functions.

```
def forward_propagation(x, theta):
    """
    Implement the linear forward propagation (compute J) presented in

    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well

    Returns:
    J -- the value of function J, computed using the formula J(theta)
    """

    ### START CODE HERE ### (approx. 1 line)
    J = np.dot(theta,x)
    ### END CODE HERE ###

    return J
```

backward_propagation(x, theta)

Exercise: Now, implement the backward propagation step (derivative computation) of Figure 1. That is, compute the derivative of $J(\theta) = \theta x$ with respect to θ. To save you from doing the calculus, you should get $dtheta = \frac{\partial J}{\partial \theta} = x$.

```
def backward_propagation(x, theta):
    """
    Computes the derivative of J with respect to theta (see Figure 1).

    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well

    Returns:
    dtheta -- the gradient of the cost with respect to theta
    """

    ### START CODE HERE ### (approx. 1 line)
    dtheta = forward_propagation(x, theta)/theta
    ### END CODE HERE ###

    return dtheta
```

gradient_check(x, theta, epsilon = 1e-7):

Exercise: To show that the `backward_propagation()` function is correctly computing the gradient $\frac{\partial J}{\partial \theta}$, let's implement gradient checking.

Instructions:

- First compute "gradapprox" using the formula above (1) and a small value of ε. Here are the Steps to follow:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$
- Then compute the gradient using backward propagation, and store the result in a variable "grad"
- Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2} \quad (2)$$

  You will need 3 Steps to compute this formula:
  - 1'. compute the numerator using np.linalg.norm(...)
  - 2'. compute the denominator. You will need to call np.linalg.norm(...) twice.
  - 3'. divide them.
- If this difference is small (say less than 10^{-7}), you can be quite confident that you have computed your gradient correctly. Otherwise, there may be a mistake in the gradient computation.

```python
def gradient_check(x, theta, epsilon = 1e-7):
    """
    Implement the backward propagation presented in Figure 1.

    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well
    epsilon -- tiny shift to the input to compute approximated gradient with formula(1)

    Returns:
    difference -- difference (2) between the approximated gradient and the backward propagation gradient
    """

    # Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need to worry about the limit.
    ### START CODE HERE ### (approx. 5 lines)
    thetaplus = theta+epsilon                               # Step 1
    thetaminus = theta-epsilon                              # Step 2
    J_plus = forward_propagation(x,thetaplus)                                  # Step 3
    J_minus = forward_propagation(x,thetaminus)                                 # Step 4
    gradapprox = (J_plus-J_minus)/(2*epsilon)                              # Step 5
    ### END CODE HERE ###

    # Check if gradapprox is close enough to the output of backward_propagation()
    ### START CODE HERE ### (approx. 1 line)
    grad = backward_propagation(x,theta)
    ### END CODE HERE ###

    ### START CODE HERE ### (approx. 1 line)
    numerator = np.linalg.norm(grad-gradapprox)                                # Step 1'
    denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)                                   # Step 2'
    difference = numerator/denominator                          # Step 3'
    ### END CODE HERE ###
```

结果：正确

```
x, theta = 2, 4
difference = gradient_check(x, theta)
print("difference = " + str(difference))
```

executed in 2ms, finished 10:19:22 2021-10-07

```
The gradient is correct!
difference = 2.919335883291695e-10
```

Expected Output: The gradient is correct!
** difference ** 2.9193358103083e-10

扩展到 N 维：

1.3 3) N-dimensional gradient checking

The following figure describes the forward and backward propagation of your fraud detection model.

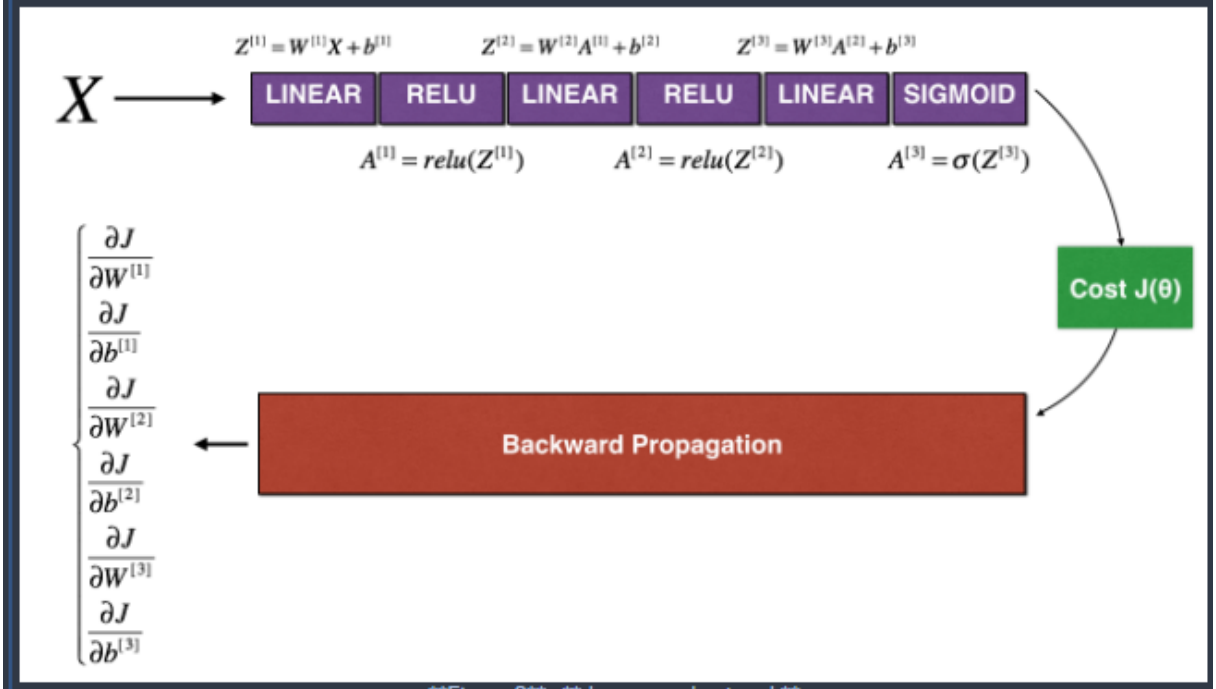


Figure 2: **deep neural network**
LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

Let's look at your implementations for forward propagation and backward propagation.

补充代码：根据图片补充代码

`gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7)`:

Exercise: Implement gradient_check_n().

Instructions: Here is pseudo-code that will help you implement the gradient check.

For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set θ^+ to `np.copy(parameters_values)`
  2. Set θ_i^+ to $\theta_i^+ + \varepsilon$
  3. Calculate J_i^+ using to `forward_propagation_n(x, y, vector_to_dictionary(` θ^+ `))`.
- To compute `J_minus[i]`: do the same thing with θ^-
- Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2} \quad (3)$$

```
# Compute gradapprox
for i in range(num_parameters):

    # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output = "J_plus[i]".
    # "_" is used because the function you have to outputs two parameters but we only care about the first one
    ### START CODE HERE ### (approx. 3 lines)
    thetaplus = np.copy(parameters_values)                                      # Step 1
    thetaplus[i][0] = thetaplus[i][0]+epsilon                                   # Step 2
    J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaplus))                                  # Step 3
    ### END CODE HERE ###

    # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output = "J_minus[i]".
    ### START CODE HERE ### (approx. 3 lines)
    thetaminus = np.copy(parameters_values)                                     # Step 1
    thetaminus[i][0] = thetaminus[i][0]-epsilon                                 # Step 2
    J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaminus))                                 # Step 3
    ### END CODE HERE ###

    # Compute gradapprox[i]
    ### START CODE HERE ### (approx. 1 line)
    gradapprox[i] = (J_plus[i]-J_minus[i])/(2*epsilon)
    ### END CODE HERE ###

# Compare gradapprox to backward propagation gradients by computing difference.
### START CODE HERE ### (approx. 1 line)
numerator = np.linalg.norm(gradapprox-grad)                                     # Step 1'
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)                                          # Step 2'
difference = numerator/denominator                                  # Step 3'
### END CODE HERE ###
```

结果：出现错误

```
In [23]:
X, Y, parameters = gradient_check_n_test_case()

cost, cache = forward_propagation_n(X, Y, parameters)
gradients = backward_propagation_n(X, Y, cache)
difference = gradient_check_n(parameters, gradients, X, Y)
```

executed in 17ms, finished 09:02:19 2021-10-07

```
There is a mistake in the backward propagation! difference = 0.28509315648437444
```

Expected output:

** There is a mistake in the backward propagation!** difference = 0.285093156781

发现 backward_propagation_n 有些错误，更改后：

```
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

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

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T)#*2
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T)
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)#4->1

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                 "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```

更改后的结果：正确

```
In [13]:
    X, Y, parameters = gradient_check_n_test_case()

    cost, cache = forward_propagation_n(X, Y, parameters)
    gradients = backward_propagation_n(X, Y, cache)
    difference = gradient_check_n(parameters, gradients, X, Y)
```

executed in 13ms, finished 09:15:07 2021-10-07

```
Your backward propagation works perfectly fine! difference = 1.1872208949889946e-07
```

Note

- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- Gradient Checking, at least as we've presented it, doesn't work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

Congrats, you can be confident that your deep learning model for fraud detection is working correctly! You can even use this to convince your CEO. :)

What you should remember from this notebook: - Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation). - Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.

1 Optimization Methods

Until now, you've always used Gradient Descent to update the parameters and minimize the cost. In this notebook, you will learn more advanced optimization methods that can speed up learning and perhaps even get you to a better final value for the cost function. Having a good optimization algorithm can be the difference between waiting days vs. just a few hours to get a good result.

1.1 1 - Gradient Descent

A simple optimization method in machine learning is gradient descent (GD). When you take gradient steps with respect to all m examples on each step, it is also called Batch Gradient Descent.

Warm-up exercise: Implement the gradient descent update rule. The gradient descent rule is, for $l = 1, \ldots, L$:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{1}$$

$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{2}$$

where L is the number of layers and α is the learning rate. All parameters should be stored in the `parameters` dictionary. Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are $W^{[1]}$ and $b^{[1]}$. You need to shift `l` to `l+1` when coding.

补充代码：根据图中的公式 1、2 补充代码即可，注意脚标

```
update_parameters_with_gd(parameters, grads, learning_rate):
```

```python
def update_parameters_with_gd(parameters, grads, learning_rate):
    """
    Update parameters using one step of gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters to be updated:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients to update each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    learning_rate -- the learning rate, scalar.

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    L = len(parameters) // 2 # number of layers in the neural networks

    # Update rule for each parameter
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate*grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate*grads["db" + str(l+1)]
        ### END CODE HERE ###

    return parameters
```

结果：正确

```
W1 = [[ 1.63535156 -0.62320365 -0.53718766]
 [-1.07799357  0.85639907 -2.29470142]]
b1 = [[ 1.74604067]
 [-0.75184921]]
W2 = [[ 0.32171798 -0.25467393  1.46902454]
 [-2.05617317 -0.31554548 -0.3756023 ]
 [ 1.1404819  -1.09976462 -0.1612551 ]]
b2 = [[-0.88020257]
 [ 0.02561572]
 [ 0.57539477]]
```

Expected Output:

W1	[[1.63535156 -0.62320365 -0.53718766] [-1.07799357 0.85639907 -2.29470142]]
b1	[[1.74604067] [-0.75184921]]
W2	[[0.32171798 -0.25467393 1.46902454] [-2.05617317 -0.31554548 -0.3756023] [1.1404819 -1.09976462 -0.1612551]]
b2	[[-0.88020257] [0.02561572] [0.57539477]]

1.2 2 - Mini-Batch Gradient descent

Let's learn how to build mini-batches from the training set (X, Y).

There are two steps:

- **Shuffle**: Create a shuffled version of the training set (X, Y) as shown below. Each column of X and Y represents a training example. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the i^{th} column of X is the example corresponding to the i^{th} label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.

Exercise: Implement `random_mini_batches`. We coded the shuffling part for you. To help you with the partitioning step, we give you the following code that selects the indexes for the 1^{st} and 2^{nd} mini-batches:

```
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 *
mini_batch_size]
...
```

Note that the last mini-batch might end up smaller than `mini_batch_size=64`. Let $\lfloor s \rfloor$ represents s rounded down to the nearest integer (this is `math.floor(s)` in Python). If the total number of examples is not a multiple of `mini_batch_size=64` then there will be $\lfloor \frac{m}{mini\_batch\_size} \rfloor$ mini-batches with a full 64 examples, and the number of examples in the final mini-batch will be $(m - mini\_batch\_size \times \lfloor \frac{m}{mini\_batch\_size} \rfloor)$.

补充代码：根据图片实现代码

`random_mini_batches(X, Y, mini_batch_size = 64, seed = 0)`:

```
# Step 1: Shuffle (X, Y)
permutation = list(np.random.permutation(m))
shuffled_X = X[:, permutation]
shuffled_Y = Y[:, permutation].reshape((1,m))

# Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini batches of size mini_batch_size in your partitionning
for k in range(0, num_complete_minibatches):
    ### START CODE HERE ### (approx. 2 lines)
    mini_batch_X = shuffled_X[:,mini_batch_size*(k):mini_batch_size*(k+1)]
    mini_batch_Y = shuffled_Y[:,mini_batch_size*(k):mini_batch_size*(k+1)]
    ### END CODE HERE ###
    mini_batch = (mini_batch_X, mini_batch_Y)
    mini_batches.append(mini_batch)

# Handling the end case (last mini-batch < mini_batch_size)
if m % mini_batch_size != 0:
    ### START CODE HERE ### (approx. 2 lines)
    mini_batch_X = shuffled_X[:,mini_batch_size*num_complete_minibatches:]
    mini_batch_Y = shuffled_Y[:,mini_batch_size*num_complete_minibatches:]
    ### END CODE HERE ###
    mini_batch = (mini_batch_X, mini_batch_Y)
    mini_batches.append(mini_batch)
```

结果：正确

```
shape of the 1st mini_batch_X: (12288, 64)
shape of the 2nd mini_batch_X: (12288, 64)
shape of the 3rd mini_batch_X: (12288, 20)
shape of the 1st mini_batch_Y: (1, 64)
shape of the 2nd mini_batch_Y: (1, 64)
shape of the 3rd mini_batch_Y: (1, 20)
mini batch sanity check: [ 0.90085595 -0.7612069   0.2344157 ]
```

Expected Output:

shape of the 1st mini_batch_X	(12288, 64)
shape of the 2nd mini_batch_X	(12288, 64)
shape of the 3rd mini_batch_X	(12288, 20)
shape of the 1st mini_batch_Y	(1, 64)
shape of the 2nd mini_batch_Y	(1, 64)
shape of the 3rd mini_batch_Y	(1, 20)
mini batch sanity check	[0.90085595 -0.7612069 0.2344157]

What you should remember: - Shuffling and Partitioning are the two steps required to build mini-batches - Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

1.3 3 - Momentum

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations.

Momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable v. Formally, this will be the exponentially weighted average of the gradient on previous steps.

You can also think of v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.

Exercise: Initialize the velocity. The velocity, v, is a python dictionary that needs to be initialized with arrays of zeros. Its keys are the same as those in the `grads` dictionary, that is: for $l = 1, \ldots, L$:

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

Note that the iterator l starts at 0 in the for loop while the first parameters are v["dW1"] and v["db1"] (that's a "one" on the superscript). This is why we are shifting l to l+1 in the `for` loop.

补充代码：根据图片实现代码

```
initialize_velocity(parameters):
```

```python
def initialize_velocity(parameters):
    """
    Initializes the velocity as a python dictionary with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the corresponding gradients/parameters.
    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl

    Returns:
    v -- python dictionary containing the current velocity.
                    v['dW' + str(l)] = velocity of dWl
                    v['db' + str(l)] = velocity of dbl
    """

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}

    # Initialize velocity
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        v["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
        v["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
        ### END CODE HERE ###

    return v
```

结果：正确

```
v["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
```

Expected Output:

v["dW1"]	[[0. 0. 0.] [0. 0. 0.]]
v["db1"]	[[0.] [0.]]
v["dW2"]	[[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
v["db2"]	[[0.] [0.] [0.]]

补充代码：根据图片实现代码

update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):

Exercise: Now, implement the parameters update with momentum. The momentum update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases} \tag{3}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases} \tag{4}$$

where L is the number of layers, β is the momentum and α is the learning rate. All parameters should be stored in the `parameters` dictionary. Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are $W^{[1]}$ and $b^{[1]}$ (that's a "one" on the superscript). So you will need to shift `l` to `l+1` when coding.

```
L = len(parameters) // 2 # number of layers in the neural networks

# Momentum update for each parameter
for l in range(L):

    ### START CODE HERE ### (approx. 4 lines)
    # compute velocities
    v["dW" + str(l+1)] = beta*v["dW" + str(l+1)]+(1-beta)*grads['dW' + str(l+1)]
    v["db" + str(l+1)] = beta*v["db" + str(l+1)]+(1-beta)*grads['db' + str(l+1)]
    # update parameters
    parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate*v["dW" + str(l+1)]
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate*v["db" + str(l+1)]
    ### END CODE HERE ###

return parameters, v
```

结果：正确

```
W1 = [[ 1.62544598 -0.61290114 -0.52907334]
 [-1.07347112  0.86450677 -2.30085497]]
b1 = [[ 1.74493465]
 [-0.76027113]]
W2 = [[ 0.31930698 -0.24990073  1.4627996 ]
 [-2.05974396 -0.32173003 -0.38320915]
 [ 1.13444069 -1.0998786  -0.1713109 ]]
b2 = [[-0.87809283]
 [ 0.04055394]
 [ 0.58207317]]
v["dW1"] = [[-0.11006192  0.11447237  0.09015907]
 [ 0.05024943  0.09008559 -0.06837279]]
v["db1"] = [[-0.01228902]
 [-0.09357694]]
v["dW2"] = [[-0.02678881  0.05303555 -0.06916608]
 [-0.03967535 -0.06871727 -0.08452056]
 [-0.06712461 -0.00126646 -0.11173103]]
v["db2"] = [[0.02344157]
 [0.16598022]
 [0.07420442]]
```

Expected Output:

W1	[[1.62544598 -0.61290114 -0.52907334] [-1.07347112 0.86450677 -2.30085497]]
b1	[[1.74493465] [-0.76027113]]
W2	[[0.31930698 -0.24990073 1.4627996] [-2.05974396 -0.32173003 -0.38320915] [1.13444069 -1.0998786 -0.1713109]]
b2	[[-0.87809283] [0.04055394] [0.58207317]]
v["dW1"]	[[-0.11006192 0.11447237 0.09015907] [0.05024943 0.09008559 -0.06837279]]
v["db1"]	[[-0.01228902] [-0.09357694]]
v["dW2"]	[[-0.02678881 0.05303555 -0.06916608] [-0.03967535 -0.06871727 -0.08452056] [-0.06712461 -0.00126646 -0.11173103]]
v["db2"]	[[0.02344157] [0.16598022] [0.07420442]]

1.4 4 - Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial J}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_1)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

As usual, we will store all parameters in the `parameters` dictionary

Exercise: Initialize the Adam variables v, s which keep track of the past information.

Instruction: The variables v, s are python dictionaries that need to be initialized with arrays of zeros. Their keys are the same as for `grads`, that is: for $l = 1, \ldots, L$:

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
s["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
s["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

补充代码：根据图片实现代码

`initialize_adam(parameters)` ：

```
L = len(parameters) // 2 # number of layers in the neural networks
v = {}
s = {}

# Initialize v, s. Input: "parameters". Outputs: "v, s".
for l in range(L):
### START CODE HERE ### (approx. 4 lines)
    v["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
    v["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
    s["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
    s["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
### END CODE HERE ###

return v, s
```

结果：正确

```
v["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
s["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
s["db1"] = [[0.]
 [0.]]
s["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
s["db2"] = [[0.]
 [0.]
 [0.]]
```

Expected Output:

v["dW1"]	[[0. 0. 0.] [0. 0. 0.]]
v["db1"]	[[0.] [0.]]
v["dW2"]	[[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
v["db2"]	[[0.] [0.] [0.]]
s["dW1"]	[[0. 0. 0.] [0. 0. 0.]]
s["db1"]	[[0.] [0.]]
s["dW2"]	[[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
s["db2"]	[[0.] [0.] [0.]]

Exercise: Now, implement the parameters update with Adam. Recall the general update rule is, for $l = 1, \ldots, L$:

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

Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are $W^{[1]}$ and $b^{[1]}$. You need to shift `l` to `l+1` when coding.

补充代码：根据图片实现代码

```
update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,
```

beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8)：

```
# Perform Adam update on all parameters
for l in range(L):
    # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
    ### START CODE HERE ### (approx. 2 lines)
    v["dW" + str(l+1)] = beta1*v["dW" + str(l+1)]+(1-beta1)*grads['dW' + str(l+1)]
    v["db" + str(l+1)] = beta1*v["db" + str(l+1)]+(1-beta1)*grads['db' + str(l+1)]
    ### END CODE HERE ###

    # Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_corrected".
    ### START CODE HERE ### (approx. 2 lines)
    v_corrected["dW" + str(l+1)] = v["dW" + str(l+1)]/(1-beta1**t)
    v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-beta1**t)
    ### END CODE HERE ###

    # Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
    ### START CODE HERE ### (approx. 2 lines)
    s["dW" + str(l+1)] = beta2*s["dW" + str(l+1)]+(1-beta2)*((grads['dW' + str(l+1)])**2)
    s["db" + str(l+1)] = beta2*s["db" + str(l+1)]+(1-beta2)*((grads['db' + str(l+1)])**2)
    ### END CODE HERE ###

    # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected".
    ### START CODE HERE ### (approx. 2 lines)
    s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)]/(1-beta2**t)
    s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-beta2**t)
    ### END CODE HERE ###

    # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output: "parameters".
    ### START CODE HERE ### (approx. 2 lines)
    parameters["W" + str(l+1)] = parameters["W" + str(l+1)]-learning_rate*v_corrected["dW" + str(l+1)]/(np.sqrt(s_corrected["dW" + str(l+1)])+epsilon)
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)]-learning_rate*v_corrected["db" + str(l+1)]/(np.sqrt(s_corrected["db" + str(l+1)])+epsilon)
    ### END CODE HERE ###
```

结果：正确

```
W1 = [[ 1.63178673 -0.61919778 -0.53561312]
 [-1.08040999  0.85796626 -2.29409733]]
b1 = [[ 1.75225313]
 [-0.75376553]]
W2 = [[ 0.32648046 -0.25681174  1.46954931]
 [-2.05269934 -0.31497584 -0.37661299]
 [ 1.14121081 -1.09244991 -0.16498684]]
b2 = [[-0.88529979]
 [ 0.03477238]
 [ 0.57537385]]
v["dW1"] = [[-0.11006192  0.11447237  0.09015907]
 [ 0.05024943  0.09008559 -0.06837279]]
v["db1"] = [[-0.01228902]
 [-0.09357694]]
v["dW2"] = [[-0.02678881  0.05303555 -0.06916608]
 [-0.03967535 -0.06871727 -0.08452056]
 [-0.06712461 -0.00126646 -0.11173103]]
v["db2"] = [[0.02344157]
 [0.16598022]
 [0.07420442]]
s["dW1"] = [[0.00121136 0.00131039 0.00081287]
 [0.0002525  0.00081154 0.00046748]]
s["db1"] = [[1.51020075e-05]
 [8.75664434e-04]]
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]]
s["db2"] = [[5.49507194e-05]
 [2.75494327e-03]
 [5.50629536e-04]]
```

Expected Output:

W1	[[1.63178673 -0.61919778 -0.53561312] [-1.08040999 0.85796626 -2.29409733]]
b1	[[1.75225313] [-0.75376553]]
W2	[[0.32648046 -0.25681174 1.46954931] [-2.05269934 -0.31497584 -0.37661299] [1.14121081 -1.09245036 -0.16498684]]
b2	[[-0.88529978] [0.03477238] [0.57537385]]
v["dW1"]	[[-0.11006192 0.11447237 0.09015907] [0.05024943 0.09008559 -0.06837279]]
v["db1"]	[[-0.01228902] [-0.09357694]]
v["dW2"]	[[-0.02678881 0.05303555 -0.06916608] [-0.03967535 -0.06871727 -0.08452056] [-0.06712461 -0.00126646 -0.11173103]]
v["db2"]	[[0.02344157] [0.16598022] [0.07420442]]
s["dW1"]	[[0.00121136 0.00131039 0.00081287] [0.0002525 0.00081154 0.00046748]]
s["db1"]	[[1.51020075e-05] [8.75664434e-04]]
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]]
s["db2"]	[[5.49507194e-05] [2.75494327e-03] [5.50629536e-04]]

最后我们实现用不同优化方法的模型：

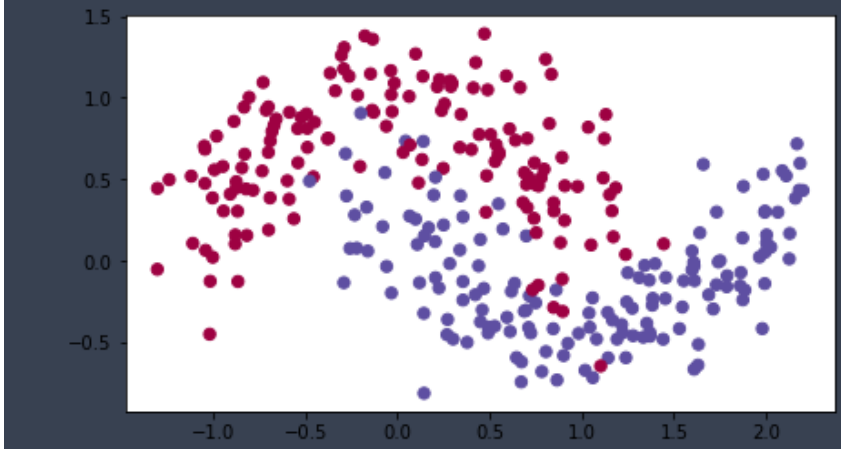
查看数据：

1.5 5 - Model with different optimization algorithms

Lets use the following "moons" dataset to test the different optimization methods. (The dataset is named "moons" because the data from each of the two classes looks a bit like a crescent-shaped moon.)

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

executed in 126ms, finished 09:49:58 2021-10-07



分别用三个优化方法训练一个三层的神经网络：

We have already implemented a 3-layer neural network. You will train it with:

- Mini-batch **Gradient Descent**: it will call your function:
  - `update_parameters_with_gd()`
- Mini-batch **Momentum**: it will call your functions:
  - `initialize_velocity()` and `update_parameters_with_momentum()`
- Mini-batch **Adam**: it will call your functions:
  - `initialize_adam()` and `update_parameters_with_adam()`

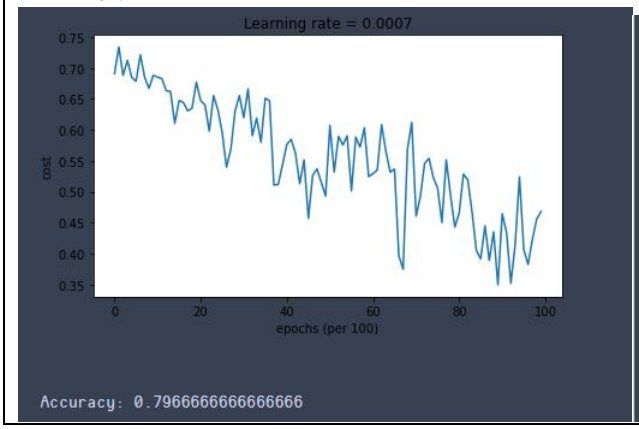
第一个：Mini-batch GD

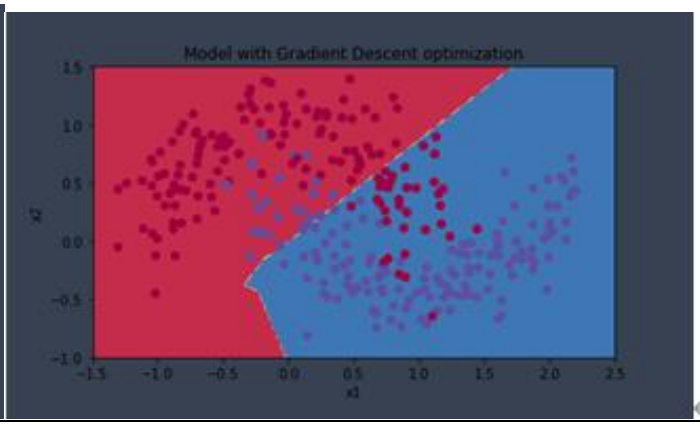
You will now run this 3 layer neural network with each of the 3 optimization methods.

1.5.1 5.1 - Mini-batch Gradient descent

Run the following code to see how the model does with mini-batch gradient descent.

结果可视化：



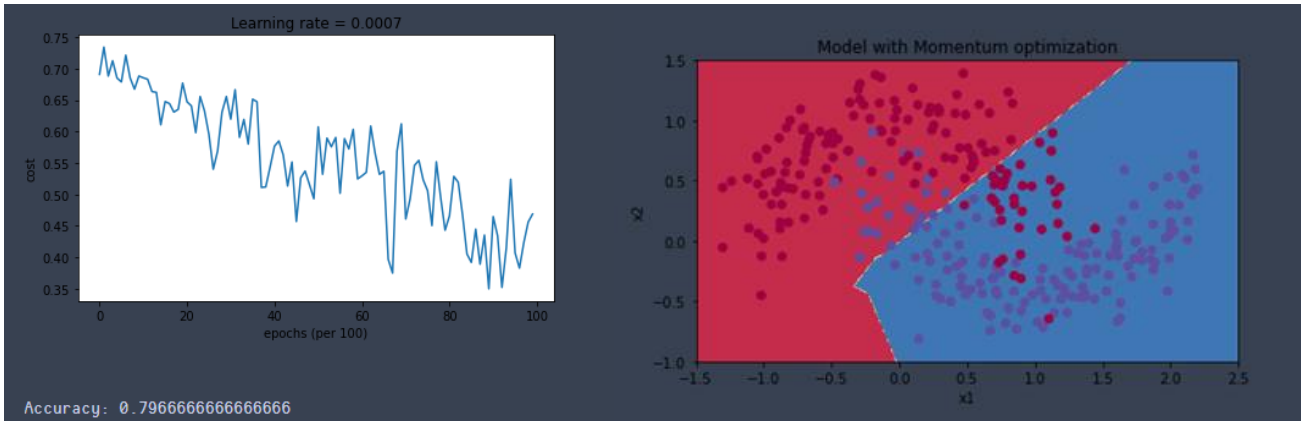


可以看到准确率 79.7%。
第二个：Mini-batch GD with momentum

1.5.2 5.2 - Mini-batch gradient descent with momentum

Run the following code to see how the model does with momentum. Because this example is relatively simple, the gains from using momemtum are small; but for more complex problems you might see bigger gains.

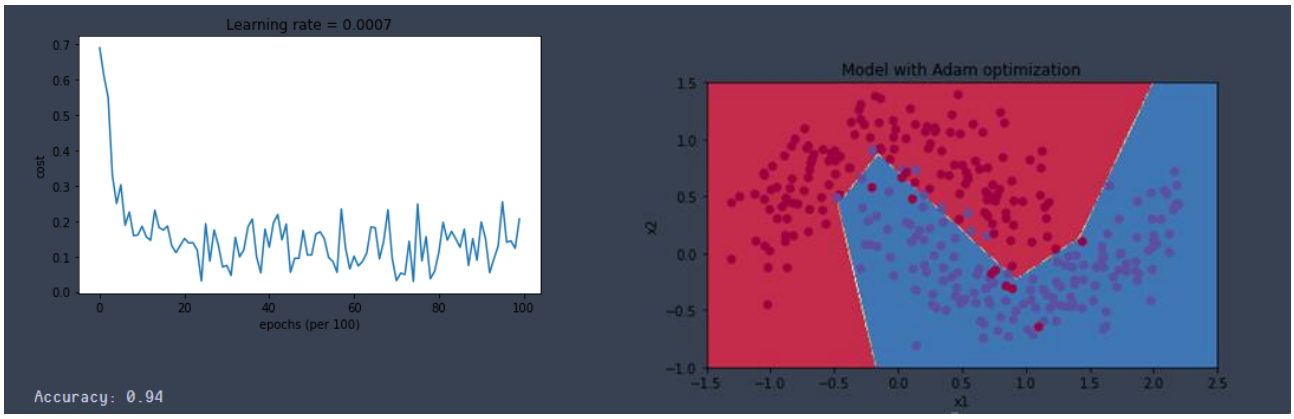
结果可视化：



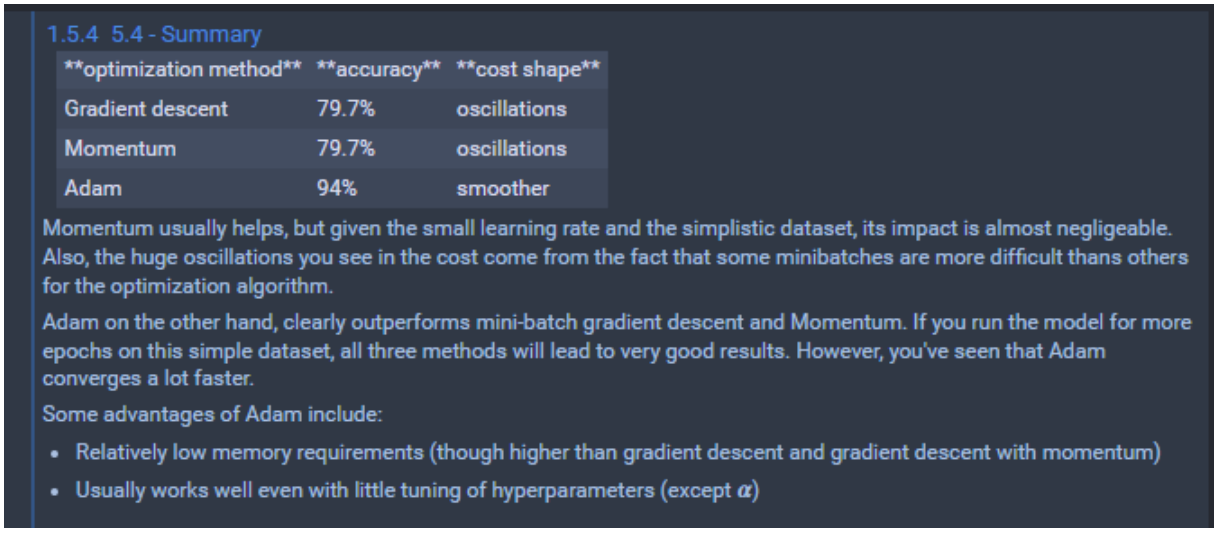
可以看到与 GD 一样，都是 79.7%，训练过程也一样。
第三个：Mini-batch with Adam mode

1.5.3 5.3 - Mini-batch with Adam mode

Run the following code to see how the model does with Adam.



可以看到不仅准确率最高，达到了 94%，而且收敛最快。



1.5.4 5.4 - Summary

optimization method	**accuracy**	**cost shape**
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

Momentum usually helps, but given the small learning rate and the simplistic dataset, its impact is almost negligeable. Also, the huge oscillations you see in the cost come from the fact that some minibatches are more difficult thans others for the optimization algorithm.

Adam on the other hand, clearly outperforms mini-batch gradient descent and Momentum. If you run the model for more epochs on this simple dataset, all three methods will lead to very good results. However, you've seen that Adam converges a lot faster.

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except α)

可见 GD 和 Momentum 得出的准确度都是 79.7%，Adam 是 94%最高，而且收敛最快。

结论分析与体会：

1， 我们希望权重初始化后既不会增长过快，也不会太快下降到 0，从而训练出一个权重或梯度不会增长或消失过快的深度网络。在训练深度网络时，这也是一个加快训练速度的技巧。

2， 梯度检验帮我们节省了很多时间，能帮我们发现 BP 实施过程中的错误，不过不要在训练中使用梯度检验，它只用于调试。

3， 梯度检验不能与 dropout 同时使用，因为每次迭代过程中，dropout 会随机消除隐藏层单元的不同子集，难以计算 dropout 在梯度下降上的代价函数。因此 dropout 可作为优化代价函数，的一种方法，但是代价函数 J 被定义为对所有指数极大的节点子集求和。而在任何迭代过程中，这些节点都有可能被消除，所以很难计算代价函数。

4， 全梯度下降计算梯度根据整个数据集合。

优点：梯度是在全部数据集上计算出的，因此每次迭代都是向着整体的最优化方向。

缺点： 1、每更新一次参数，就要遍历全部数据集，计算起来非常慢。 2、容易陷入极小值点，因为在极小值点（鞍点）梯度为 0，所以参数不会更新。

5，SGD（随机梯度下降）与 BGD 一次更新用所有的数据计算不同，SGD 每次更新时对随机选择一个样本进行梯度更新。

优点：

与全梯度下降相比，更新参数时速度快。与全梯度相比，SGD 可能会跳出局部极小值点，因为在极小值（鞍点）的时候它计算梯度是随机选择的一个样本，这个梯度未必是 0

缺点：

SGD 每次的更新并不是向着整体最优化方向，虽然速度快，准确度下降，并不是全局最优。虽然具有随机性，但是从期望上看，它是等于正确的导数。

6，Mini-Batch Gradient Descent 每次使用一小批数据更新，是全梯度和随机梯度的一个折衷。

缺点：

不能保证很好的收敛性，依赖学习率。如何学习率太小，收敛慢，如果太大就会在极小值附近震荡。batch-size 大小的选择很重要。

7，Momentum 与梯度下降不同的是在计算梯度地方，当前时刻的梯度是 从开始时刻到当前时刻的梯度指数加权平均，并给这个梯度的指数加权值取了个名字速率 v， 既有方向也有大小。

优点：

与梯度下降相比，下降速度快，因为如果方向是一直下降的，那么速度将是之前梯度的和，

所以比仅用当前梯度下降快。
对于窄长的等梯度图，会减轻梯度下降的震荡程度，因为考虑了当前时刻是考虑了之前的梯度方向，加快收敛

8，Adam:(Adaptive Moment Estimation)，上面的算法中的学习率都是固定不变的，Adam 是对学习率自适应调整。
Adam 不仅存储了过去梯度的平方的指数衰减平均值，还向 Monentum 一样保持了过去提取的指数衰减平均值
优点：
学习率自适应修正，不用手动调整.
自适应学习率可以对低频的参数做较大的更新，对高频的做较小的更新

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

1， 在写代码的时候一开始没有看 markdown 里的提示，然后就会各种错误，各种不知道后来仔细阅读后发现补充代码就是把 markdown 里面的公式或者伪代码实现,问题非常简单的解决。

2， 一开始没有理解梯度检验的原理，后来经过查资料后明白，数值检验估计梯度的方法类似于求数值导数的方式。对梯度的估计采用的方法是在代价函数上沿着切线的方向选择离两个非常近的点然后计算两个点的平均值用以估计梯度。即对于某个特定的 θ，我们计算出在 θ－ε 处和 θ＋ε 的代价值，然后求两个代价的平均，用以估计在 θ 处的代价值。当 θ 是一个向量时，我们则需要对偏导数进行检验。最后我们还需要与通过反向传播方法计算出的偏导数进行检验对比。梯度检测方法的开销是非常大的，比反向传播算法的开销都大，所以一旦用梯度检测方法确认了梯度下降算法算出的梯度（或导数）值是正确的，那么就及时关闭它。

3， 深度理解 Momentum：为了解决随机梯度下降的优化曲线过于曲折和速度较慢的缺点，我们可以引入动量。动量算法积累了之前梯度的指数加权平均，并且建议继续想这个方向移动，对于下一次的梯度更新有指导作用。由于动量积累了前一刻的梯度方向，所以当下一次的梯度更新方向与之相反时，受前一刻相反梯度的影响，当前时刻的梯度变化幅度会减小；反之会增强。形象的说，可以把动量想象成物理中的势能。当一个跑步的人想要右拐的时候，由于势能的影响，他不可能 90 度垂直拐弯，取而代之的是一个不那么锋利的圆滑弧线。

4，深度理解 Adam：Adam 算法和传统的随机梯度下降不同。随机梯度下降保持单一的学习率（即 alpha）更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率 Adam 算法的提出者描述其为两种随机梯度下降扩展式的优点集合，即：适应性梯度算法（AdaGrad）为每一个参数保留一个学习率以提升在稀疏梯度（即自然语言和计算机视觉问题）上的性能。均方根传播（RMSProp）基于权重梯度最近量级的均值为每一个参数适应性地保留学习率。这意味着算法在非稳态和在线问题上有很有优秀的性能。Adam 算法同时获得了 AdaGrad 和 RMSProp 算法的优点。Adam 不仅如 RMSProp 算法那样基于一阶矩均值计算适应性参数学习率，它同时还充分利用了梯度的二阶矩均值（即有偏方差）。具体来说，算法计算了梯度的指数移动均值，超参数 beta1 和 beta2 控制了这些移动均值的衰减率。移动均值的初始值和 beta1、beta2 值接近于 1（推荐值），因此矩估计的偏差接近于 0。该偏差通过首先计算带偏差的估计而后计算偏差修正后的估计而得到提升。