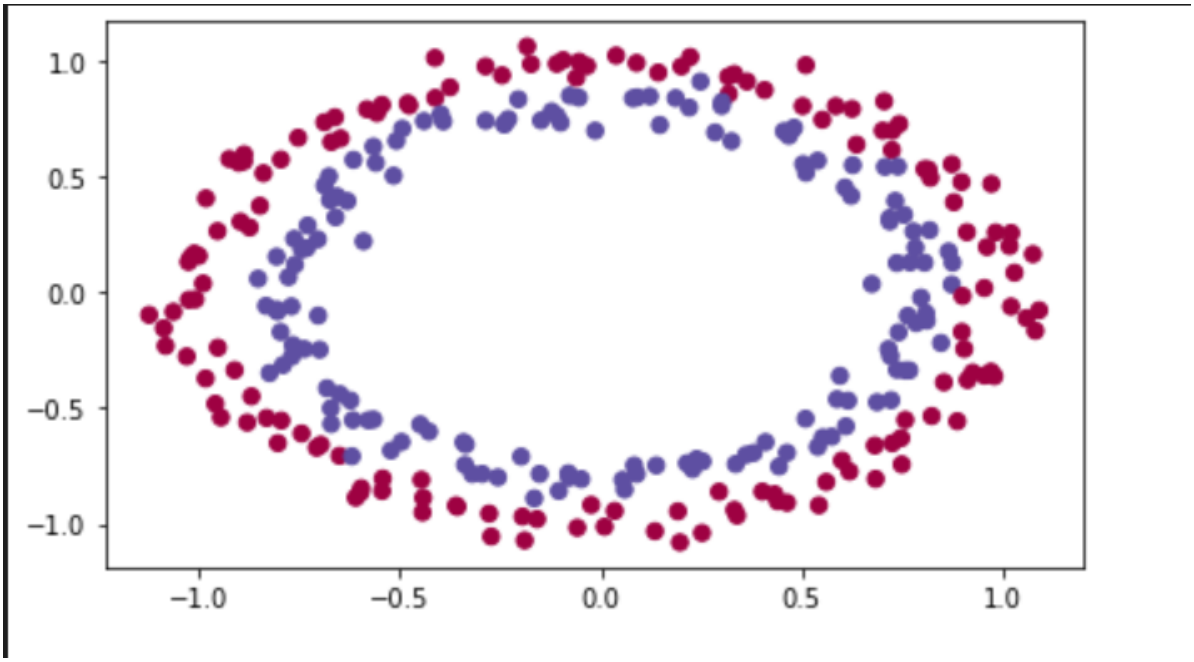
# 计算机科学与技术学院神经网络与深度学习课程实验报告

| | | |
|---|---|---|
| 实验题目：实验 2_1 | | 学号：201900161098 |
| 日期：10.4 | 班级： 智能 19 | 姓名： 马田慧 |
| Email：tianhuima01@gmail.com | | |
| 实验目的：<br>图像分类，knn，svm，softmax，3 层神经网络已经相应的向量化实现 | | |
| 实验软件和硬件环境：<br>Dell，jupyter notebook | | |
| 实验原理和方法：<br>Hyperparameter tuning, Regularization and Optimization | | |
| 实验步骤：（不要求罗列完整源代码）<br>按照实验指导书完成相应内容，查阅相关资料；<br>总结提升模型的参数。<br>具体记录在后面给出表格后给出。 | | |
| 结论分析与体会：<br>本实验从 jupyter notebook 中提取有效内容，认识到模型的初始化，梯度检查，最优化的方法与重要性。 | | |
| 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：<br>1. 不理解 adam<br>参照公式写代码，查阅资料<br>2. 报告提交格式不能满足笔记积累需求，md 记录主要内容，然后 pdf 拼接。 | | |

# Initialization

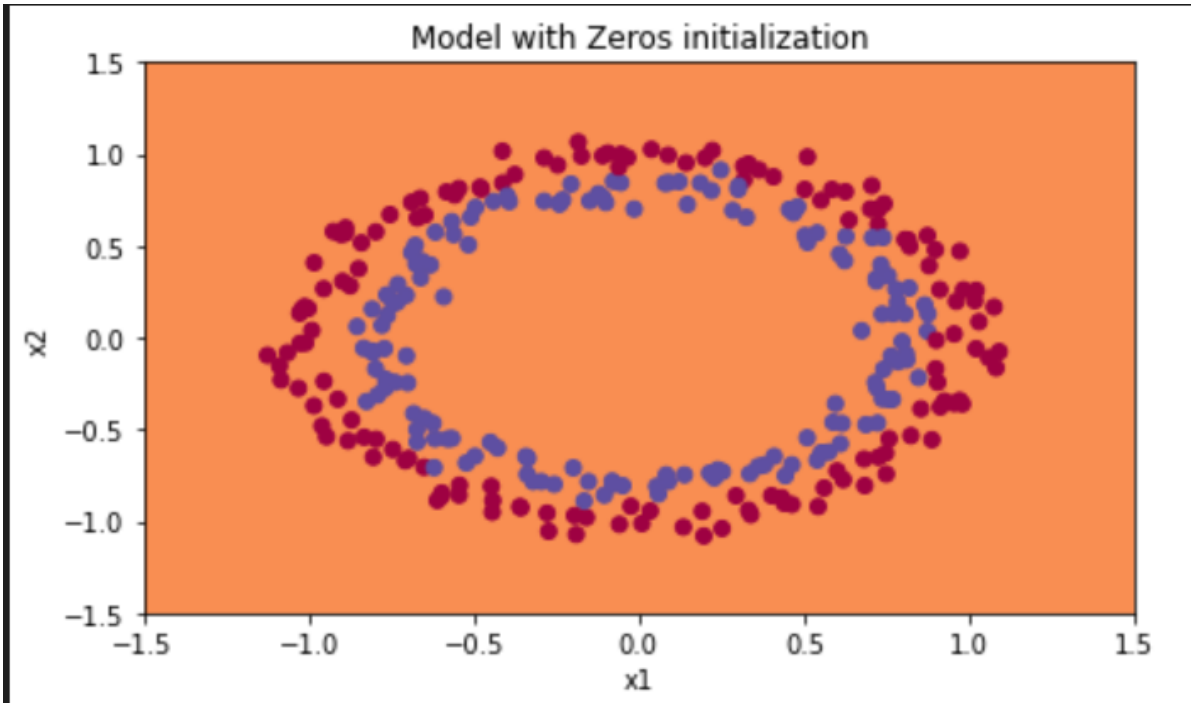
## Zero initialization

原始数据分布：




w=0

结果输出只有一个类别不做区分：




统一颜色表示同一个类别；

explain：

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.

**conclusion:**

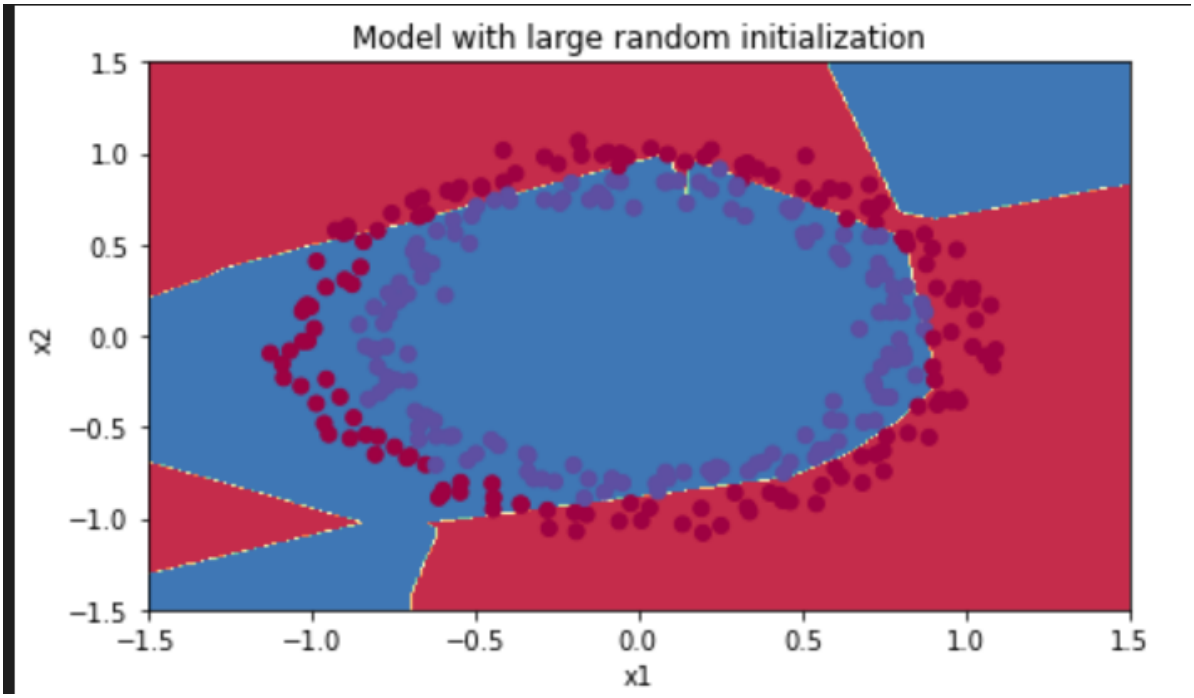
**The weights $W^{[l]}$ should be initialized randomly to break symmetry.**

**- It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.**

## Random initialization

```
        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))
```

results:

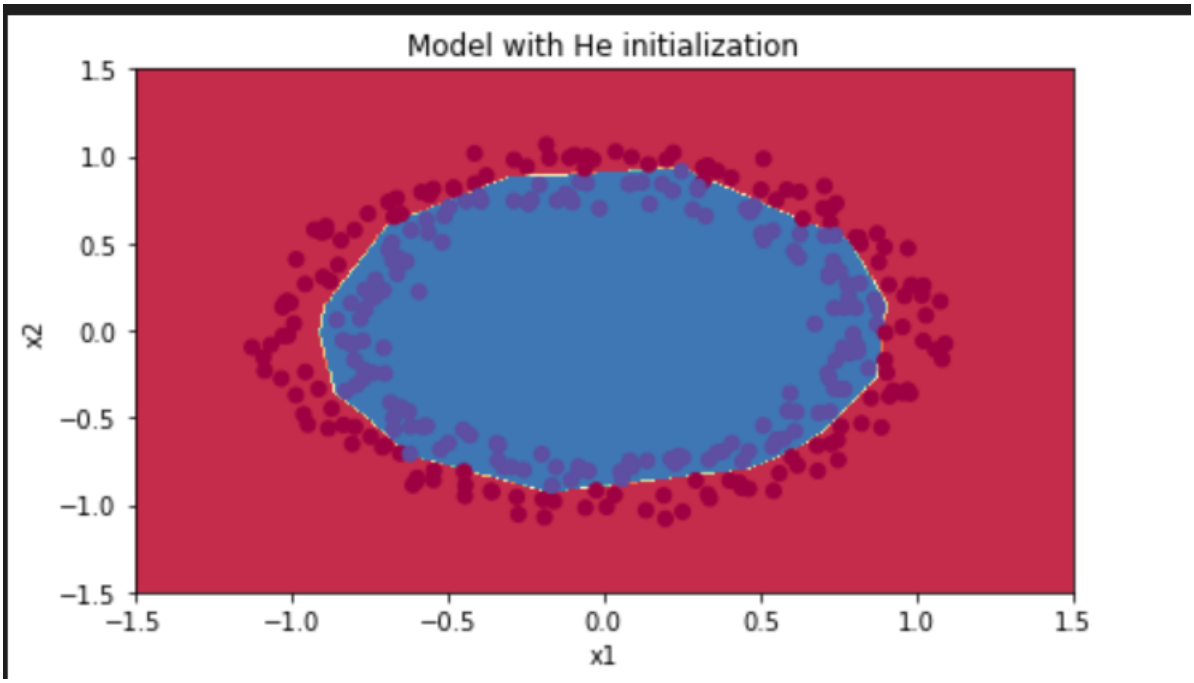



## He initialization

论文中给出的最好的初始化方法;

```
        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))
```

results:




## conclusion

**-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 intialize to values that are too large**

**- He initialization works well for networks with ReLU activations.**

# Gradient Checking

reassurance: proof that your backpropagation is actually working

## numerick check

$$difference = \frac{||grad - gradapprox||_2}{||grad||_2 + ||gradapprox||_2} \tag{2}$$

`np.linalg.norm()`:计算x的l2距离

```
  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(gradapprox-grad)                                           #
Step 1'
    denominator = np.linalg.norm(gradapprox)+np.linalg.norm(grad)
            # Step 2'
    difference = numerator/denominator
```

> sigmod derivition: $s'(x) = s(x) \cdot (1 - s(x))$

## Conclusion

- 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.

# Optimization Methods

## Gradient Descent

一次选择所有上的梯度进行更新

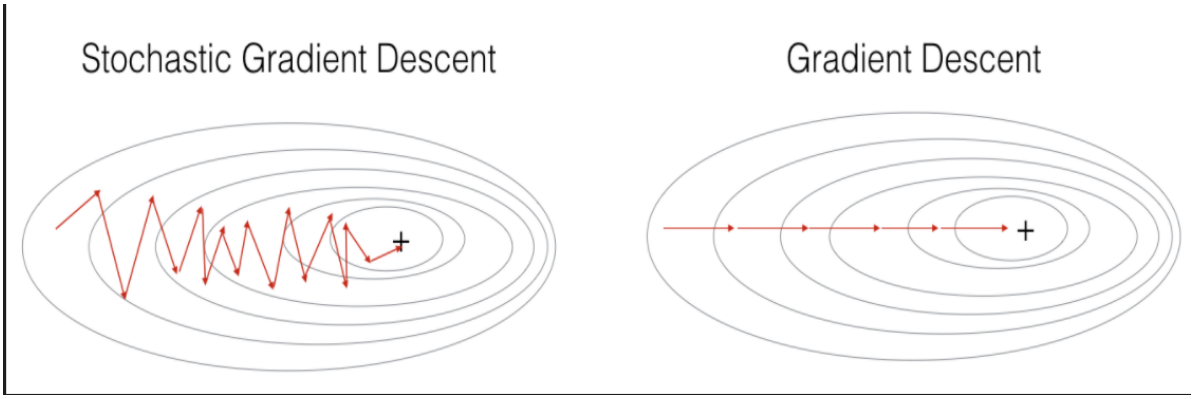
$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \qquad (1)$$

$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \qquad (2)$$

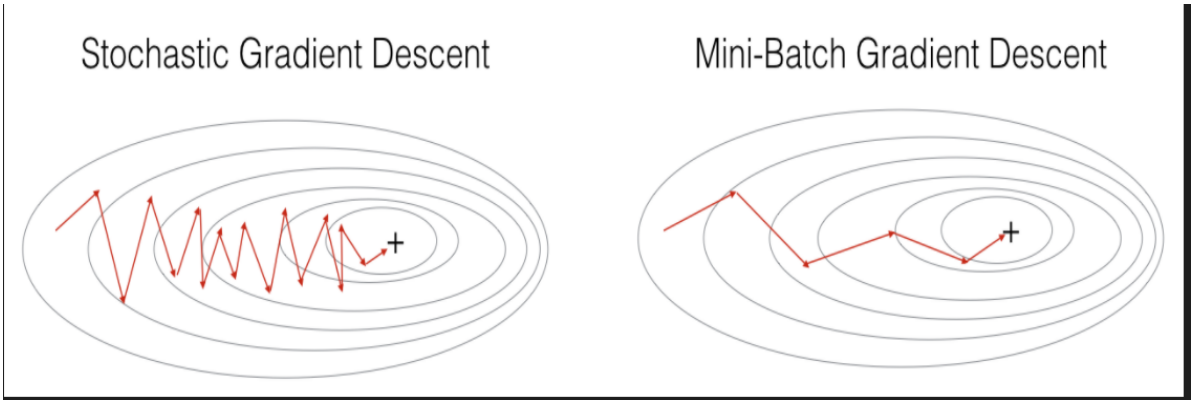
## SGD

一次选择一个的梯度进行更新

```
X = data_input

Y = labels

parameters = initialize_parameters(layers_dims)

for i in range(0, num_iterations):

  \# Forward propagation

  a, caches = forward_propagation(X, parameters)

  \# Compute cost.

  cost = compute_cost(a, Y)

  \# Backward propagation.

  grads = backward_propagation(a, caches, parameters)

  \# Update parameters.

  parameters = update_parameters(parameters, grads)
```




## Mini-Batch Gradient denscent




随机打乱排序，分成小批量

打乱：

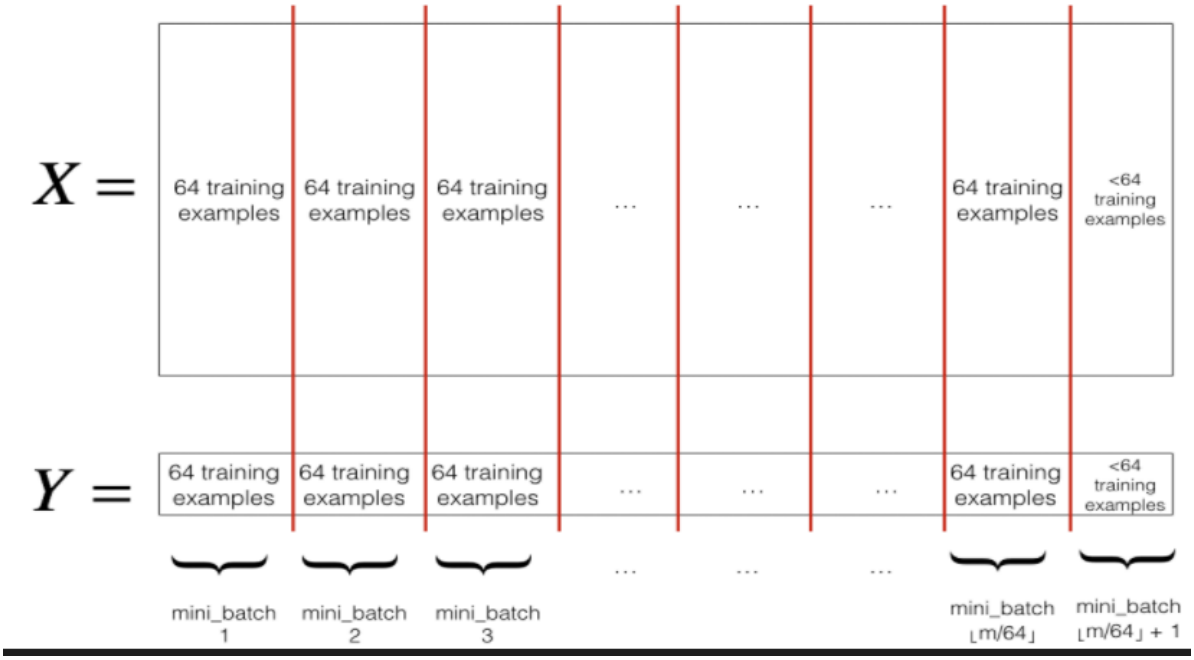
$$X=\begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \dots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)}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

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

$$Y=\begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$

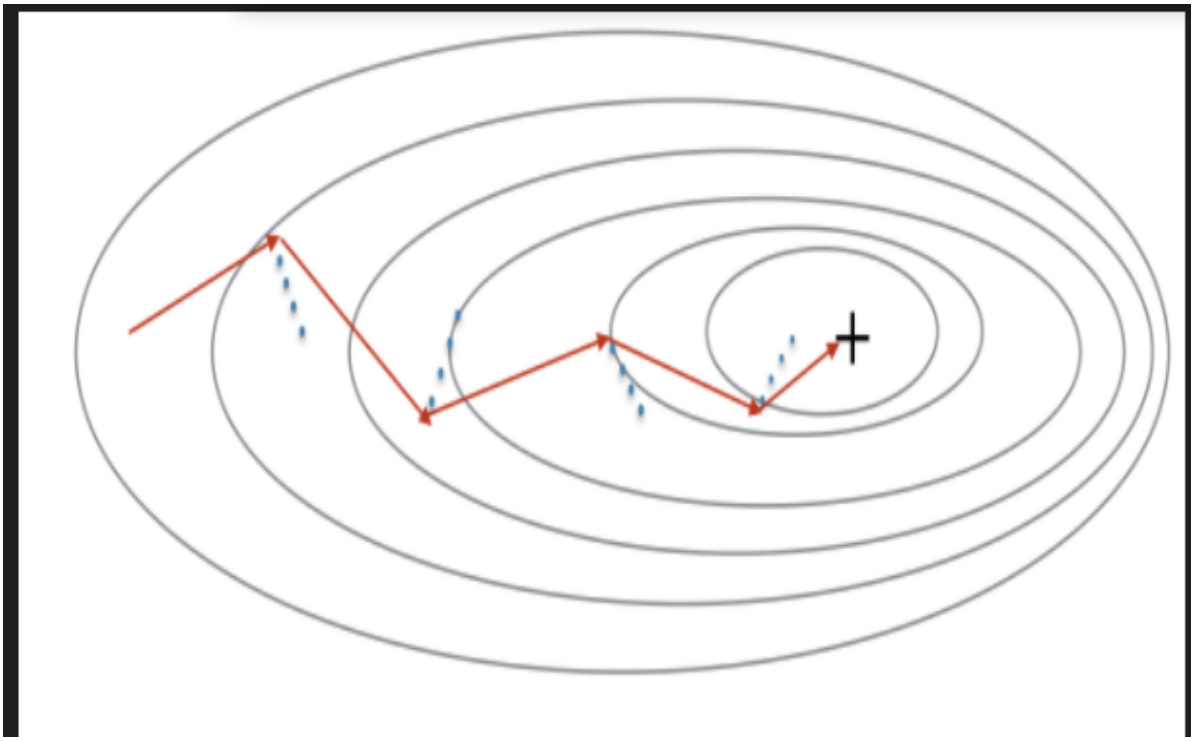
$$Y=\begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$

分批量：




## Momentum

Momentum takes into account the past gradients to smooth out the update.



```
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)])
```

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

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

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

# Choose $\beta$?*

Common values for $\beta$ range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.

# Conclusion

Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent

# Adam

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".

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

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

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

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

## 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 $\alpha$)