暑假学习记录

7.18(主要总结傅里叶变化及其应用)

markdowm的基本语法

简单学习了markdowm的语法，例如**加粗的文字**，*倾斜的文字*，***倾斜加粗的文字***，~~加删除线的文字~~

有序列表

1. 我有一个梦想
2. 我有两个梦想
3. 我有三个梦想

无序列表

- 使用【星号】标识无序列表
- 使用【星号】标识无序列表
- 使用【星号】标识无序列表

链接网址，若是图片只需在最前面加一个!

```
printf("Hello world")
```

```
# 这是在markdown中插入的python代码块
for i in range(10)
print i
```

傅里叶变换及其滤波应用

傅里叶变化

傅里叶变换理论部分 傅里叶变换，我的理解就是将一幅图像从时域变换到其对应的频域进行分析，而在时域不好进行滤波处理，但在频域就相对好进行滤波处理。 至于什么是傅里叶变化，什么是时域，什么是频域，可以看看这篇，确实没学过数学的也能看懂，不过也太生活化了。 如果看了这篇文章你还不懂傅里叶变换，那就过来掐死我吧 至于数学操作方面，如果这让我讲讲也太难为我了。

代码参考：

```
    Mat padded;                     //以0填充输入图像矩阵
    int m = getOptimalDFTSize(I.rows);
    int n = getOptimalDFTSize(I.cols);

    //填充输入图像I，输入矩阵为padded，上方和左方不做填充处理
```

```
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols,
BORDER_CONSTANT, Scalar::all(0));

    Mat planes[] = { Mat_<float>(padded), Mat::zeros(padded.size(),CV_32F)
};
    Mat complexI;
    merge(planes, 2, complexI);     //将planes融合合并成一个多通道数组complexI

    dft(complexI, complexI);        //进行傅里叶变换

    //计算幅值，转换到对数尺度(logarithmic scale)
    //=> log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
    split(complexI, planes);        //planes[0] = Re(DFT(I),planes[1] =
Im(DFT(I))
                                    //即planes[0]为实部,planes[1]为虚部
    magnitude(planes[0], planes[1], planes[0]);     //planes[0] = magnitude
    Mat magI = planes[0];

    magI += Scalar::all(1);
    log(magI, magI);                //转换到对数尺度(logarithmic scale)

    //如果有奇数行或列，则对频谱进行裁剪
    magI = magI(Rect(0, 0, magI.cols&-2, magI.rows&-2));

    //重新排列傅里叶图像中的象限，使得原点位于图像中心
    int cx = magI.cols / 2;
    int cy = magI.rows / 2;

    Mat q0(magI, Rect(0, 0, cx, cy));       //左上角图像划定ROI区域
    Mat q1(magI, Rect(cx, 0, cx, cy));      //右上角图像
    Mat q2(magI, Rect(0, cy, cx, cy));      //左下角图像
    Mat q3(magI, Rect(cx, cy, cx, cy));     //右下角图像

    //变换左上角和右下角象限
    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    //变换右上角和左下角象限
    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);

    //归一化处理，用0-1之间的浮点数将矩阵变换为可视的图像格式
    normalize(magI, magI, 0, 1, CV_MINMAX);
```

opencv学习（十五）之图像傅里叶变换dft **下面分享一下我认为困难的地方**

```
    magI += Scalar::all(1);
    log(magI, magI);
```

之所以要进行对数转换是因为傅里叶变换后的结果对于在显示器显示来讲范围比较大，这样的话对于一些小的变化或者是高的变换值不能进行观察。因此高的变化值将会转变成白点，而较小的变化值则会变成黑点。为了能够获得可视化的效果，可以利用灰度值将我们的线性尺度(linear scale)转变为对数尺度(logarithmic scale)

```
normalize(magI, magI, 0, 1, CV_MINMAX);
```

对结果进行归一化处理同样是处于可视化的目的。现在我们得到了幅度值，但是这仍然超出了0-1的显示范围。这就需要利用normalize()函数对数据进行归一化处理。 而按理说归一到（0，1），图片应该是漆黑的一片，如果我们imwrite把图片到处发现确实如我们所料，这里就是imshow的特性。如果输入图像是32位浮点型（32-bitfloating-point），像素值便要乘以255。也就是说，该值的范围是[0,1]映射到[0,255]。但为什么归一化到0-1再乘255和直接归一化在0-255的效果不同呢，这是因为取对数后的频谱依然存在大量大于255的幅值，如果直接归一化在0-255，这些大于255的数值依然是255.归一化在0~1再乘255相当于进行了缩放，先缩小范围在0-1（尽管大量的值依然集中在1附近），然后按比例放大到图像灰度值的范围。

可以看看这片博客，里面讲了傅里叶变化这两方面的内容

OpenCV中的傅里叶的门道

滤波处理

高通，低通，带通，带阻

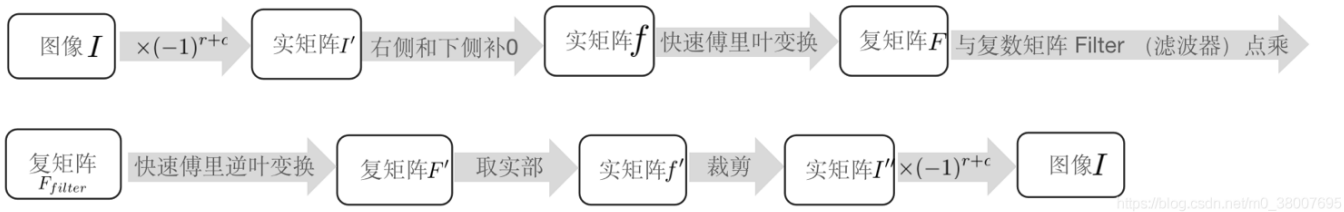
接下来了解两个重要的概念：低频和高频。低频指的是图像的傅里叶变换“中心位置”附近的区域。高频随着到“中心位置”距离的增加而增加，即傅里叶变换中心位置的外围区域，这里的“中心位置”指的是傅里叶变换所对应的幅度谱最大值的位置。

针对图像的傅里叶变换，低频信息表示图像中灰度值缓慢变化的区域；而高频区域则正好相反，表示灰度值变化迅速的部分，如边缘。低通滤波，保留傅里叶变换的低频信息，或者削弱傅里叶变换的高频信息；而高通滤波正好相反，保留傅里叶变换的高频信息，移除或者削弱傅里叶变换的低频信息。

高通就是保留了高频的信息，结果是提取了图像的边缘，低通则是过滤了高频信息，是整个图像变得更加平滑，带通滤波是指指保留某一范围区域的频率带，与带通滤波相反，带阻滤波是指撤销或消弱指定范围区域的频率带。

而滤波器实质上就是元素为0或1的矩阵，滤波器与原图像相乘，1的地方表示保留，0则变为黑色。

一个完整的滤波处理步骤如图所示:



带阻滤波器代码 下面写一个高斯带组滤波器： gbpFilter(r,c)=1-exp(-((D(r,c)^2-D0^2)/D(r,c)/BW)^2) D(r,c)中心位置的距离，D0代表带宽的径向中心,BW表示带宽

```

```

```
Mat gaussian_block_kernel(Mat src,float D0,float BW){//src为原图经过图像填充
    Mat gaussianBlur(src.size(),CV_32FC1);
    for(int i=0;i<src.rows;i++){
        for(int j=0;j<src.cols;j++){
            float d = pow(float(i - src.rows / 2), 2) + pow(float(j -
src.cols / 2), 2);//分子,计算pow必须为float型
            float f=(d*d-D0*D0)/(d*BW);
            gaussianBlur.at<float>(i,j)=1-expf(-(f*f));
        }
    }
    imshow("高斯带阻滤波器",gaussianBlur);
    return gaussianBlur;
}
```

本想给出滤波器的图像，可是导出后果然一片漆黑..... 可以参考下面这篇博客，给出各种滤波的公式

OpenCV —— 频率域滤波（傅里叶变换，低通和高通滤波，带通和带阻滤波，同态滤波）

7.19（卡尔曼滤波的小总结）

理论部分 目前来看，这个东西还是学的一知半解. 在卡尔曼滤波下，一个物体真实的状态，应该是预测量加测量量，而具体怎么加法，就看哪个量更加精确，我们就会给其更大的比例。 下面给出数学公式。

$$x_k^- = Ax_{k-1} + Bu_{k-1}$$

$$P_k^- = AP_{k-1}A^T + Q$$

$$K_k = \frac{P_k^- H^T}{HP_k^- H^T + R}$$

$$x_k = x_k^- + K_k(z_k - Hx_k^-) \qquad z_k = Hx_{k\text{测量}}$$

$$P_k = (I - K_k H)P_k^-$$

CSDN @Deepcong

x_{k-1}：第k-1 个过程的后验状态估计，就是去掉噪声的融合数据变量。

u_{k-1}：过程控制变量；

A：状态转移矩阵；

B：控制矩阵；

$\bar{x_k}$：第k个过程的先验状态估计，就是去掉噪声预测得来的状态变量；

P_{k-1}:第$k-1$个过程的后验误差协方差矩阵；

Q：过程噪声协方差矩阵；

$\bar{P_k}$：第k个过程的先验误差协方差矩阵；

H：观测矩阵；

R：观测噪声协方差矩阵；

K_k：卡尔曼增益系数；

$x_{k测量}$：第k个过程的测量状态变量；

z_k：去掉噪声的观测变量；

x_k：第k个过程的后验状态估计，就是去掉噪声的融合数据变量；

P_k：第k个过程的后验误差协方差矩阵；

CSDN @Deepcong

这些公式我也没有很能理解，只是知道了大概，感觉得学了概率论在来好好看看。 这应该是我看到过讲的最详细下比较好理解的。 从放弃到精通！卡尔曼滤波从理论到实践~ ***代码部分*** 先放上opencv中的示例:kalman.cpp

```
#include "opencv2/video/tracking.hpp"
#include "opencv2/highgui.hpp"
#include <iostream>
#include <stdio.h>
```

```
using namespace std;
using namespace cv;

//计算相对窗口的坐标值，因为坐标原点在左上角，所以sin前有个负号
static inline Point calcPoint(Point2f center, double R, double angle)
{
    return center + Point2f((float)cos(angle), (float)-sin(angle))*
(float)R;
}

static void help()
{
    printf( "\nExamle of c calls to OpenCV's Kalman filter.\n"
"   Tracking of rotating point.\n"
"   Rotation speed is constant.\n"
"   Both state and measurements vectors are 1D (a point angle),\n"
"   Measurement is the real point angle + gaussian noise.\n"
"   The real and the estimated points are connected with yellow line
segment,\n"
"   the real and the measured points are connected with red line
segment.\n"
"   (if Kalman filter works correctly,\n"
"    the yellow segment should be shorter than the red one).\n"
           "\n"
"   Pressing any key (except ESC) will reset the tracking with a different
speed.\n"
"   Pressing ESC will stop the program.\n"
           );
}

int main(int, char**)
{
    help();
    Mat img(500, 500, CV_8UC3);
    KalmanFilter KF(2, 1, 0);                                    //创建卡尔
曼滤波器对象KF
    Mat state(2, 1, CV_32F);                                     //state(角
度，△角度)
    Mat processNoise(2, 1, CV_32F);
    Mat measurement = Mat::zeros(1, 1, CV_32F);                 //定义测量值
    char code = (char)-1;

    for(;;)
    {
        //1.初始化
        randn( state, Scalar::all(0), Scalar::all(0.1) );          //
        KF.transitionMatrix = *(Mat_<float>(2, 2) << 1, 1, 0, 1);  //转移矩
阵A[1,1;0,1]



        //将下面几个矩阵设置为对角阵
        setIdentity(KF.measurementMatrix);                                //测
量矩阵H
        setIdentity(KF.processNoiseCov, Scalar::all(1e-5));            //系
```

```
统噪声方差矩阵Q
        setIdentity(KF.measurementNoiseCov, Scalar::all(1e-1));        //测
量噪声方差矩阵R
        setIdentity(KF.errorCovPost, Scalar::all(1));                  //后
验错误估计协方差矩阵P

        randn(KF.statePost, Scalar::all(0), Scalar::all(0.1));
//x(0)初始化

        for(;;)
        {
            Point2f center(img.cols*0.5f, img.rows*0.5f);          //center
图像中心点
            float R = img.cols/3.f;                                //半径
            double stateAngle = state.at<float>(0);                //跟踪点
角度
            Point statePt = calcPoint(center, R, stateAngle);     //跟踪点坐
标statePt

            //2. 预测
            Mat prediction = KF.predict();                        //计算预测
值，返回x'
            double predictAngle = prediction.at<float>(0);          //预测点
的角度
            Point predictPt = calcPoint(center, R, predictAngle);   //预测点
坐标predictPt


            //3.更新
            //measurement是测量值
            randn( measurement, Scalar::all(0),
Scalar::all(KF.measurementNoiseCov.at<float>(0)));     //给measurement赋值
N(0,R)的随机值

            // generate measurement
            measurement += KF.measurementMatrix*state;  //z = z + H*x;

            double measAngle = measurement.at<float>(0);
            Point measPt = calcPoint(center, R, measAngle);

            // plot points
            //定义了画十字的方法，值得学习下
            #define drawCross( center, color, d )
\
                line( img, Point( center.x - d, center.y - d ),
\
                             Point( center.x + d, center.y + d ), color, 1,
CV_AA, 0); \
                line( img, Point( center.x + d, center.y - d ),
\
                             Point( center.x - d, center.y + d ), color, 1,
CV_AA, 0 )

            img = Scalar::all(0);
```

```
            drawCross( statePt, Scalar(255,255,255), 3 );
            drawCross( measPt, Scalar(0,0,255), 3 );
            drawCross( predictPt, Scalar(0,255,0), 3 );
            line( img, statePt, measPt, Scalar(0,0,255), 3, CV_AA, 0 );
            line( img, statePt, predictPt, Scalar(0,255,255), 3, CV_AA, 0
);


            //调用kalman这个类的correct方法得到加入观察值校正后的状态变量值矩阵
            if(theRNG().uniform(0,4) != 0)
                KF.correct(measurement);

            //不加噪声的话就是匀速圆周运动，加了点噪声类似匀速圆周运动，因为噪声的原
因，运动方向可能会改变
            randn( processNoise, Scalar(0),
Scalar::all(sqrt(KF.processNoiseCov.at<float>(0, 0))));   //vk
            state = KF.transitionMatrix*state + processNoise;

            imshow( "Kalman", img );
            code = (char)waitKey(100);

            if( code > 0 )
                break;
        }
        if( code == 27 || code == 'q' || code == 'Q' )
            break;
    }

    return 0;
}
```

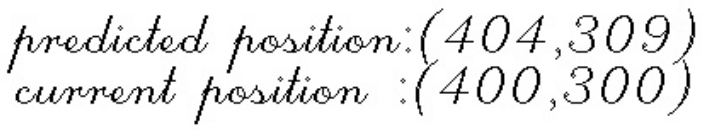
这段代码我也没很能理解。 不是很能理解state这个变量的实际意义，首先，它不可能是预测值，预测值可以由KF.predict()这个方法得到，也不是测量值，测量值是measurment,感觉也不是状态值，状态值应该是KF.statePost. 而至于各种初始化矩阵也不是很能理解，虽然知道是可以自己设置Q，R，P，X(0)的值。 下面有一段跟踪鼠标位置的代码

```
#include "opencv2/video/tracking.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/highgui/highgui_c.h>
#include <stdio.h>
#include<iostream>
using namespace cv;
using namespace std;

const int winHeight=600;
const int winWidth=800;


Point mousePosition= Point(winWidth>>1,winHeight>>1);

//mouse event callback
```

```
void mouseEvent(int event, int x, int y, int flags, void *param )
{
    if (event==CV_EVENT_MOUSEMOVE) {
        mousePosition = Point(x,y);
    }
}

int main (void)
{
    RNG rng;
    //1.kalman filter setup
    const int stateNum=4;                                    //状态值4×1向
量(x,y,△x,△y)
    const int measureNum=2;                                  //测量值2×1向
量(x,y)
    KalmanFilter KF(stateNum, measureNum, 0);

    KF.transitionMatrix = (Mat_<float>(4, 4)
<<1,0,1,0,0,1,0,1,0,0,1,0,0,0,0,1);  //转移矩阵A
    setIdentity(KF.measurementMatrix);
//测量矩阵H
    setIdentity(KF.processNoiseCov, Scalar::all(1e-5));
//系统噪声方差矩阵Q
    setIdentity(KF.measurementNoiseCov, Scalar::all(1e-1));
//测量噪声方差矩阵R
    setIdentity(KF.errorCovPost, Scalar::all(1));
//后验错误估计协方差矩阵P
    rng.fill(KF.statePost,RNG::UNIFORM,0,winHeight>winWidth?
winWidth:winHeight);   //初始状态值x(0)
    cout<<KF.statePost.at<float>(0)<<' '<<KF.statePost.at<float>(1)<<'
'<<KF.statePost.at<float>(2)<<' '<<KF.statePost.at<float>(3)<<endl;
    Mat measurement = Mat::zeros(measureNum, 1, CV_32F);
//初始测量值x'(0)，因为后面要更新这个值，所以必须先定义

    namedWindow("kalman");
    setMouseCallback("kalman",mouseEvent);

    Mat image(winHeight,winWidth,CV_8UC3,Scalar(0));

    while (1)
    {
        //2.kalman prediction
        Mat prediction = KF.predict();

        Point predict_pt = Point(prediction.at<float>
(0),prediction.at<float>(1) );   //预测值(x',y')

        //3.update measurement
        measurement.at<float>(0) = (float)mousePosition.x;
        measurement.at<float>(1) = (float)mousePosition.y;

        //4.update
        KF.correct(measurement);
        Point p=Point(KF.statePost.at<float>(0),KF.statePost.at<float>(1)
```

```
);
        //draw
        image.setTo(Scalar(255,255,255,0));
        circle(image,predict_pt,5,Scalar(0,255,0),3);    //predicted point
with green
        circle(image,mousePosition,5,Scalar(255,0,0),3); //current position
with red
        //circle(image,p,5,Scalar(0,0,255),5); //current position with red

        //cout<<KF.statePost.at<float>(0)<<' '<<KF.statePost.at<float>(1)
<<' '<<KF.statePost.at<float>(2)<<' '<<KF.statePost.at<float>(3)<<endl;

        char buf[256];
        sprintf(buf,"predicted position:
(%3d,%3d)",predict_pt.x,predict_pt.y);

putText(image,buf,Point(10,30),CV_FONT_HERSHEY_SCRIPT_COMPLEX,1,Scalar(0,0,
0),1,8);
        sprintf(buf,"current position :
(%3d,%3d)",mousePosition.x,mousePosition.y);

putText(image,buf,cvPoint(10,60),CV_FONT_HERSHEY_SCRIPT_COMPLEX,1,Scalar(0,
0,0),1,8);

        imshow("kalman", image);
        int key=waitKey(3);
        if (key==27){//esc
            break;
        }
    }
}
```

这段代码我觉得比上面的好理解，measurement是根据mouseEvent得到的鼠标测量值坐标，predict_pt就是预测值坐标，而KF.statePost就应该是状态值了。



以上可以参考这篇 [学习OpenCV2——卡尔曼滤波(KalmanFilter)详解] {https://blog.csdn.net/GDFSG/article/details/50904811} 卡尔曼滤波的总结先到这里，后面可以在做补充，继续、深入学习。

7.24（初入深度学习）

我是从b站上吴恩达的视频看其的，截至到现在，看完了第三周，做一个小小的总结。

***第二周* 建立神经网络的主要步骤是：** 定义模型结构（例如输入特征的数量） 初始化模型的参数 循环： 3.1 计算当前损失（正向传播） 3.2 计算当前梯度（反向传播） 3.3 更新参数（梯度下降） 其中比较重要的一点是多个样本的向量化，因为for循环时间消耗远远大于向量化，而这个其实就是线性代数的简单推导了。

对于正向传播 x (i)

z(i)=wTx(i)+b

y^(i)=a(i)=sigmoid(z(i))

L(a(i),y(i))=−y(i)log(a(i))−(1−y(i))log(1−a(i))

而对于反向传播：

dw = (1 / m) * np.dot(X, (A - Y).T)

db = (1 / m) * np.sum(A - Y)

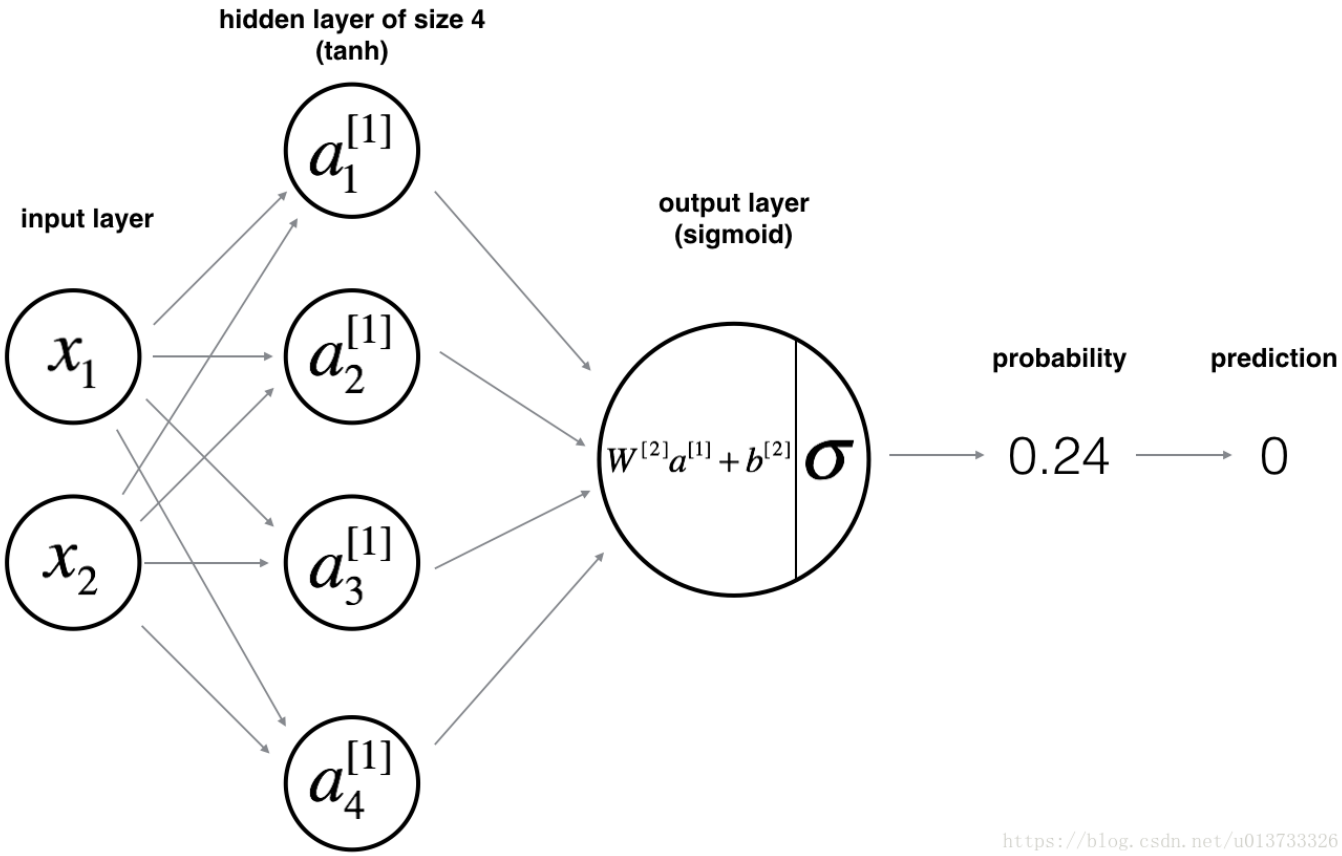
而对于更新：

w = w - learning_rate * dw

b = b - learning_rate * db

这样，一个最基本的神经网络就是这样构成的。

第三周 我们要建立一个神经网络，它有一个隐藏层。例如老师举的例子，房子的价钱并不是只和面积有关，还和许多有关，例如卧室，环境等等。因此，这里可以加一个隐藏层。 例子如下图：



其实和上一周的神经网络基本一样，多了一层隐藏层，其实也就是多了一层运算，而方法和步骤与上一周的基本是一样的。 **构建神经网络的一般方法是：**

1. 定义神经网络结构（输入单元的数量，隐藏单元的数量等）。
2. 初始化模型的参数
3. 循环 另外，需要注意的是，这里的初始化的w是不能全部初始为0的，而b可以初始为0. 老师也介绍了除了sigmoid之外的激活函数，如ranh，RELU等，而这些的效果基本都优于sigmoid，而老师也提到过他基本默认先使用RELU。 对于 x (i) x^{(i)} x(i) 而言:

z1=W[1]x(i)+b1

a1=tanh(z1)

z2=W[2]a1+b2

y^(i)=a2=σ(z2)

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]}A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(\mathrm{z}^{[1]})$	$dZ^{[1]} = W^{[2]T}dZ^{[2]} * g^{[1]\prime}(\mathrm{Z}^{[1]})$
$dW^{[1]} = dz^{[1]}x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]}X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

而对于更新，同上一周。