Neural networks and deep learning

Algorithm Presentation

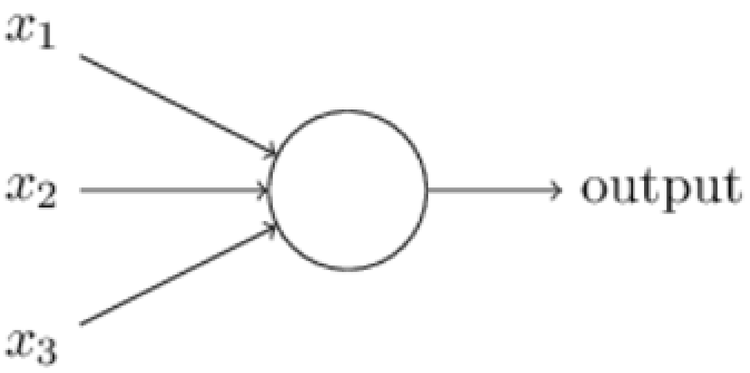
田奇

预备知识

- 多元微分
  - 链式法则
  - 梯度（随机梯度下降 not only for neural networks）
- 线性代数
  - 矩阵向量运算（高维变量运算）
- 概率统计
  - 均方差（误差评估，拟合数据not only for neural networks ）

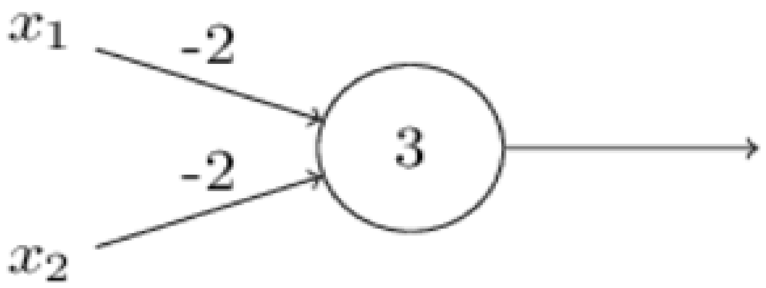
例子

- 神经网络模拟加法器
  - 最简单的网络结构
    - 感知机（Perceptrons），用于判断真假
    - output $= \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$



神经网络模拟加法器

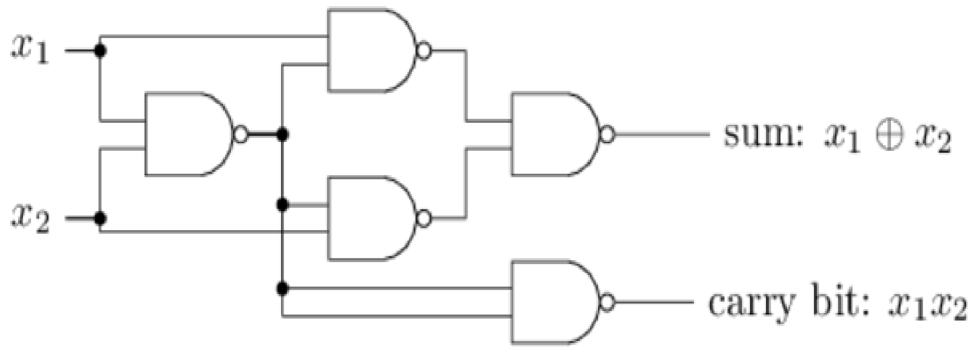
中国科学技术大学
University of Science and Technology of China

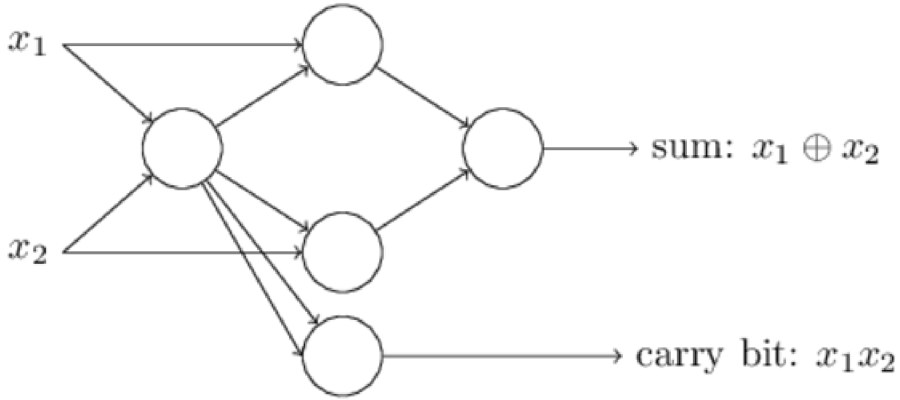
- $\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \le 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$ $w \cdot x \equiv \sum_j w_j x_j$
- 

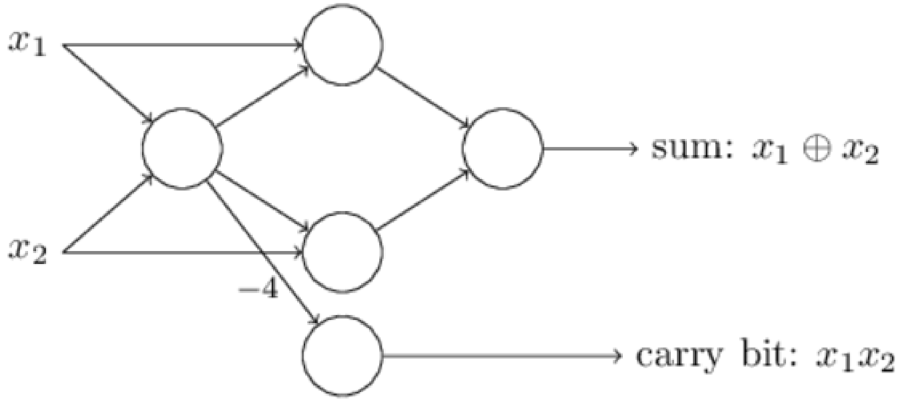
NAND与非门！！！

	0	1
0	$(-2)*0+(-2)*0+3=3$ 1	$(-2)*0+(-2)*1+3=1$ 1
1	$(-2)*1+(-2)*0+3=1$ 1	$(-2)*1+(-2)*1+3=-1$ 0

神经网络模拟加法器



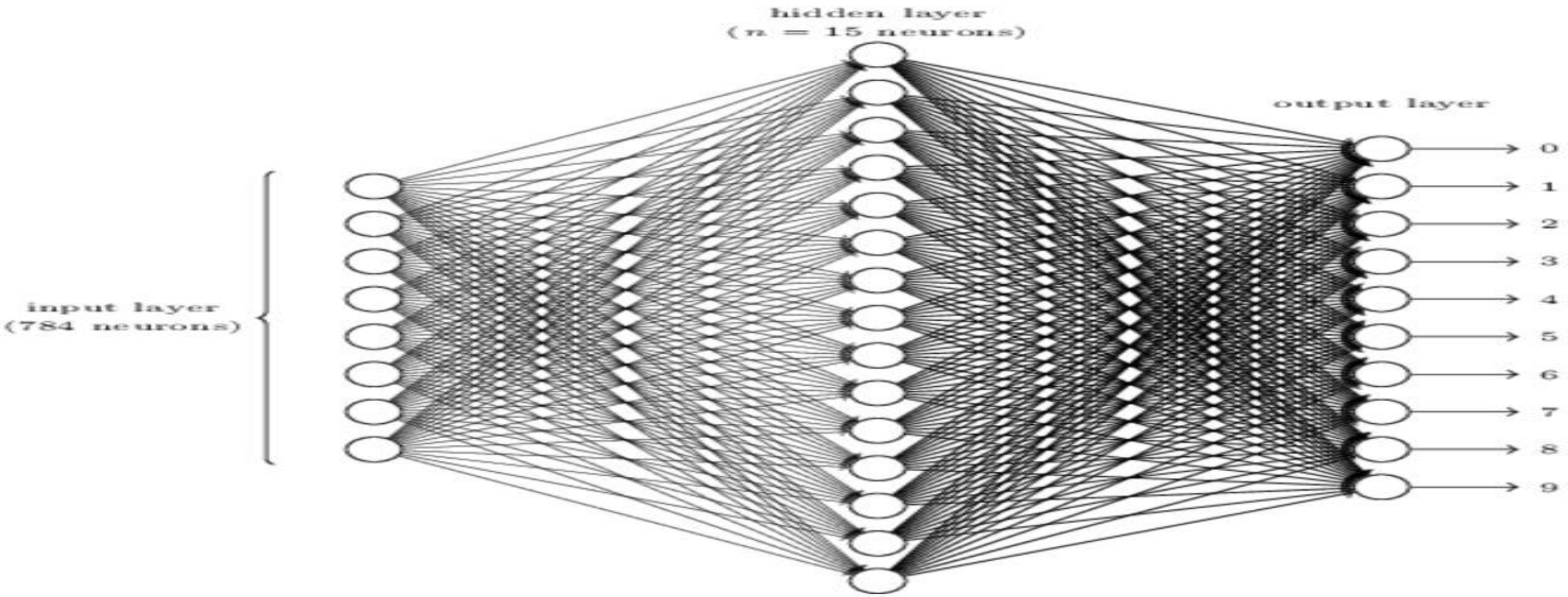


- 

Awesome！！！

启发

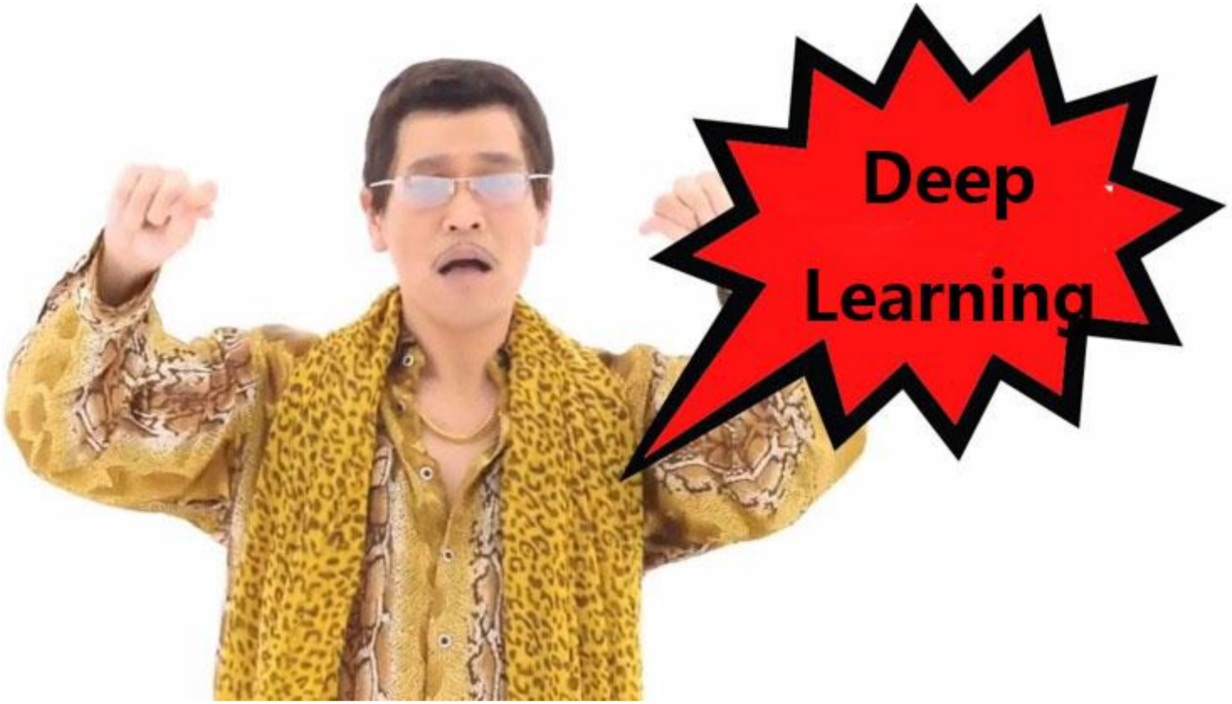
- 神经网络是否可以模拟任意的模型？
  - 理论上是可以的！



启发

- 所有的识别感知问题都可以使用神经网络来训练！！！
- 图像处理专家，语音识别专家，自然语言处理专家，人工智能专家，机器视觉专家........

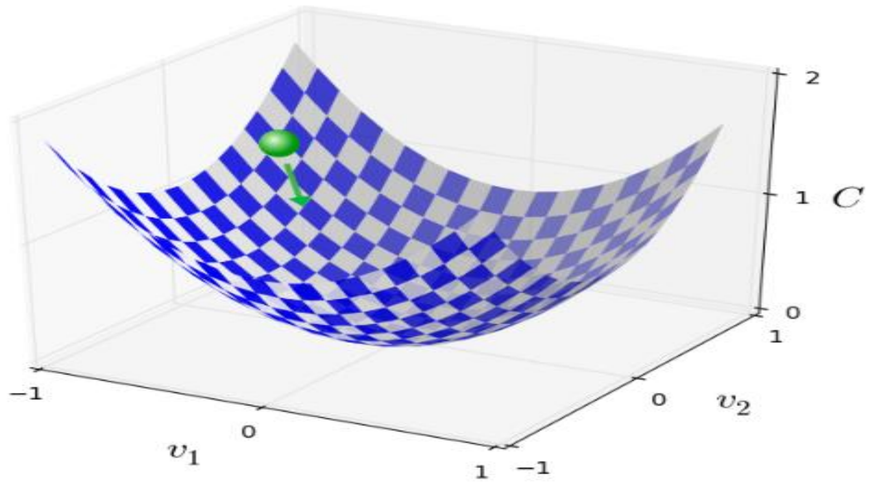
- All in one method
- Deep learning



神经网络模型

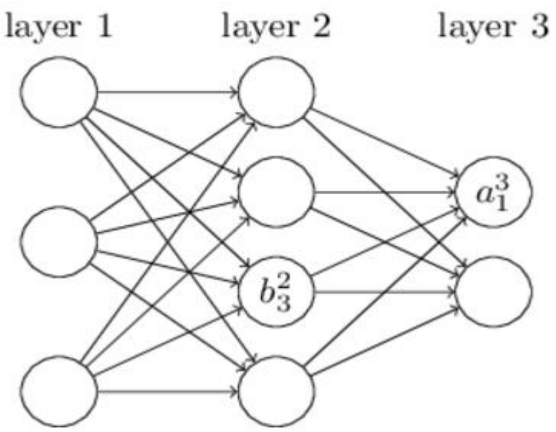
- 预热
  - 高中的线性回归
    - 假设我们有很多二维空间的样本点(x_i, y_i)，我们预判这些样本点满足一次线性方程即y $= a * x + b$.
    - 如何计算a和b。经典的是最小二乘法，即计算min C $= \sum_i (ax_i + b - y_i)^2$.
    - 计算方法分别对a和b求偏导数，令其值为0。我们可以直接解出a和b是关于x_i, y_i式子。
  - 随机梯度下降
    - 计算a b的另一种方法实际上是无限逼近，就是先给a和b设定任意值，然后根据公式 $a \overset{\Delta}{\rightarrow} a - \eta \frac{\partial C}{\partial a}$ 和 $b \overset{\Delta}{\rightarrow} b - \eta \frac{\partial C}{\partial b}$ 反复使用样本(x_i, y_i)更新a 和 b，直到拟合到我们想要的结果！

- 随机梯度下降
  - 为什么要使用 $\eta\frac{\partial C}{\partial a}$ 和 $\eta\frac{\partial C}{\partial b}$ 来更新a和b呢？假设a和b分别代表 v_1 和 v_2 . $\left(\frac{\partial C}{\partial a},\frac{\partial C}{\partial b}\right)$ 是C的梯度，我们知道二元函数C沿着梯度方向反方向是下降最快的，因此使用梯度来更新！ η 是一个学习步长，为常数！



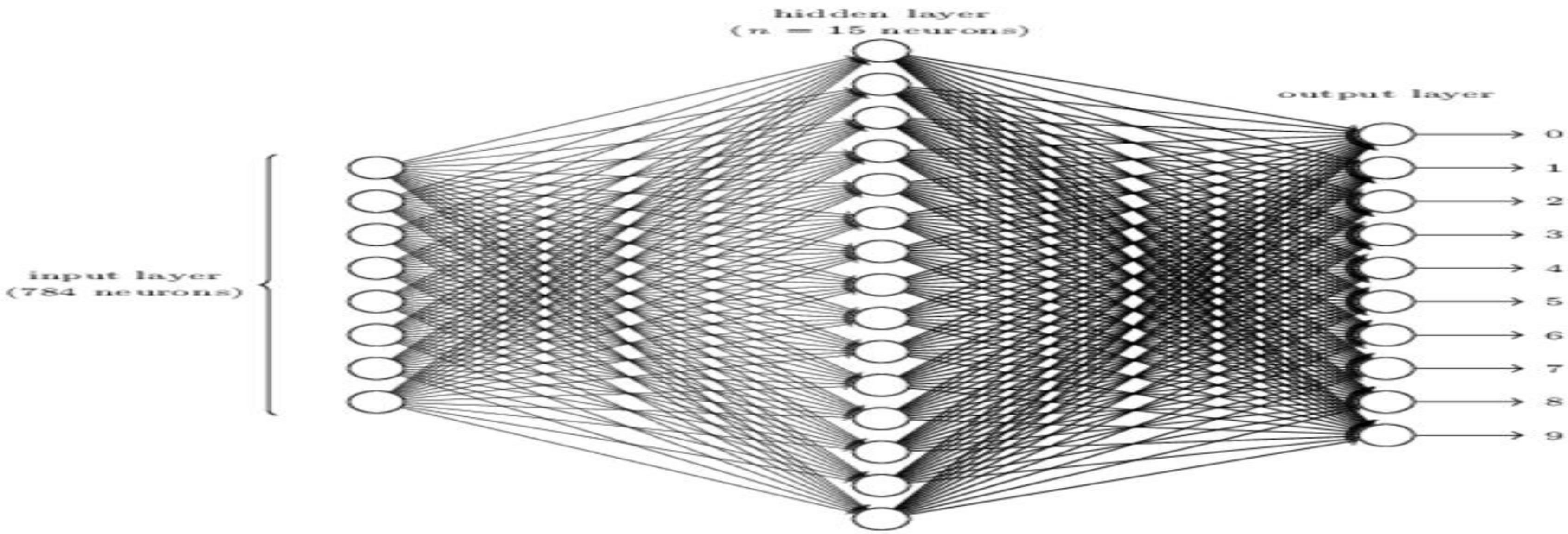
神经网络模型

- 二维空间到高维空间
  - 如果我们的样本数据变成了高维有序对即(α,β)，α,β 分别是向量。
  - 最小二乘法还能用吗?如何计算系数？
- 二层到多层
  - 如果我们的输出再作为下一层的输入继续计算，如何构建我们的模型？



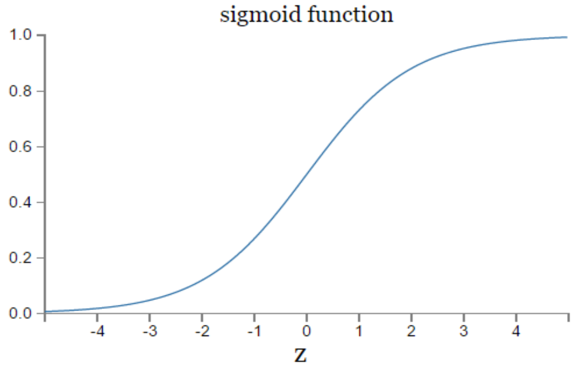
神经网络模型

- 模型架构
  - 我们把所有问题简单归结为映射，给定输入数据input，我们的网络结构直接计算，输出预测结果output.
  - 但是我们的网络结构初始状态是不成熟的，他需要学习！
  - 典型的三层神经网络结构



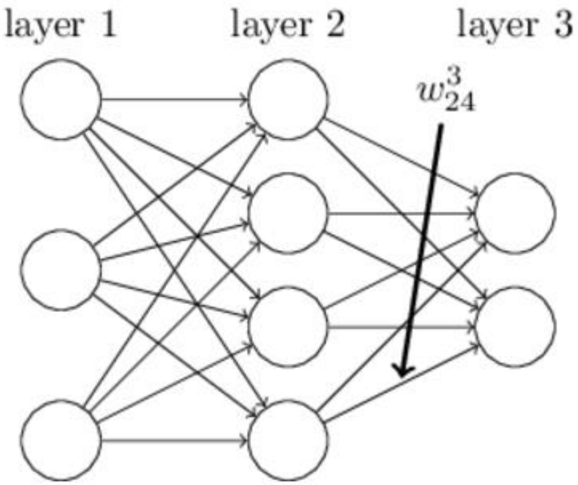
神经网络模型

- Sigmoid函数
  - 回顾前面我们的感知器 $\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \le \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$，他对于输出结构就只有两种可能，即只有两类。神经网络需要更多的输出类别，并且对细微的改变也能做出输出上的细调整。
  - 为了让神经网络能够对细微的差别做出细微的改变，我们引入一个重要的函数输出函数Sigmoid. Sigmoid函数具备很多有趣的数学特性！
    - $\sigma(z) \equiv \frac{1}{1+e^{-z}}$.
    - $z = w \cdot x + b$
    - $a = \sigma(z)$ a为每一层每一个单元的输出激活
    - $\frac{\partial a}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z)) = a(1-a)$.



神经网络模型

- 模型公式
  - 标量公式 $a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$, $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
    - 激活输出层的神经元 a_j^l 与 前一层的所有神经元 a_k^{l-1}相关
  - 向量公式 $a^l = \sigma(w^l a^{l-1} + b^l)$. $z^l \equiv w^l a^{l-1} + b^l$



w_{jk}^l is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

神经网络模型

- 模型评估（这里只讲均方差）
  - 均方差C，输入为(x, y)
    - 向量式 $C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2$,
    - 标量式 $C = \frac{1}{2n}\sum_x \sum_j (y_j^x - a_j^{x,L})^2$,
  - 训练模型的目的是使均方差最小，均方差C在偏导数全部为0的地方是最小的。（其实是局部最小，但实践表现很好）
    - 也就是说我们需要对所有的权值w和偏差b求偏导数。
    - 但是，很不好直接求出来！！多层网络结构有非常多的权值矩阵和偏差向量。

反向传播算法（BackPropagation）

- 最后一层L和L-1层之间的权值导数（链式法则）
  - 公式 $a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$, $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
    - $\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{jk}^L} = (a_j^L - y_j^L) \cdot \sigma'(z_j^L) \cdot a_k^{L-1}$
    - $\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial b_j^L} = (a_j^L - y_j^L) \cdot \sigma'(z_j^L) \cdot 1$
  - 通过对一个样本正向计算一次，即可得到 z_j^l a_j^l ,因此最后一层的权值导数是直接可求的。
  - 那么中间层的权值和偏差导数如何求解呢？
    - 为了方便，我们先定义 $\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l)$

反向传播算法（BackPropagation）

- 我们暂且把 δ_j^l 叫每一层每一个神经元的误差项，则
  - $\frac{\partial C}{\partial w_{jk}^L} = \delta_j^L \cdot a_k^{L-1}$ $\frac{\partial C}{\partial b_j^L} = \delta_j^L$ $\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l)$
- 我们真的需要计算出中间层的导数吗？要知道中间层的导数表达出来是一个很复杂的式子。
  - 递推法（链式法则）
    - $\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1}$ $\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot 1$
  - δ_j^L 是直接求出来的，那么 δ_j^l 是否可以通过 δ_j^L 递推？

反向传播算法（BackPropagation）

- 递推
  - $\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$
  - $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}.$
  - $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l).$
  - $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l).$

反向传播算法（BackPropagation）

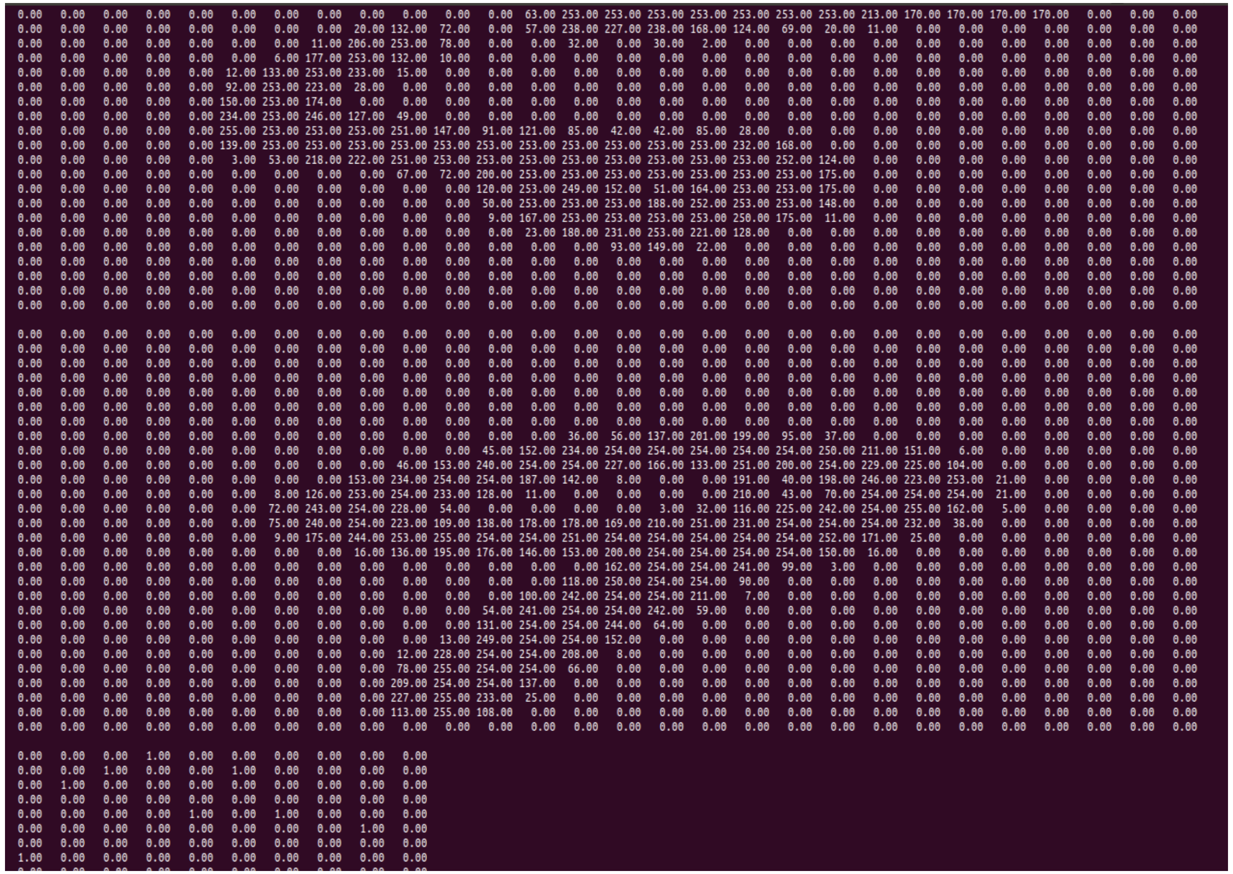
- 总结向量矩阵形式的公式
  - $\delta^L = \nabla_a C \odot \sigma'(z^L).$
  - $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l),$
  - $\frac{\partial C}{\partial b_j^l} = \delta_j^l.$
  - $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l.$
- 为什么要写成向量的形式？
  - 因为标量公式涉及大量的角标，例如 w_{jk}^l 就有三个角标，显然程序至少是三重循环，并且对每一个样本x,又是一层循环，再来最后训练多次，又是一层循环，这样写程序很难维护，角标容易错误。因此采用封装和抽象的思想，封装好矩阵运算，我们就可以直接减少三重循环。易于代码的阅读和维护。

反向传播算法（BackPropagation）

- 1.Input x: Set the corresponding activation a^1 for the input layer
- 2.Feedforward: For each l = 2,3,…,L compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
- 3.Output error δ^L:Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
- 4.Backpropagate the error:For each l = L-1,L-2,…2 compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$,
- 5.Output: The gradient of cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$. and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.
- 6.Gradient descent: For each l=L,L-1,…,2 update the weights according to the rule $w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T$ and the biases according to the rule $b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$

C语言实现版本

- 采用MNIST数据集（手写字符图片）
  - MNIST数据集是 28 * 28的灰度图片，60000个训练样例和10000个测试样例



C语言实现版本

FOLDERS
- demo
  - mnist-parser
  - matrix.c
  - matrix.h
  - mnist_reader.c
  - mnist_reader.h
  - nn
  - nn.c
  - nn1.c
  - test_images.txt
  - test_labels.txt
  - test_matrix
  - test_matrix.c
  - test_reader
  - test_reader.c
  - train_images.txt
  - train_labels.txt

nn.c

```c
}

int SGD(matrix *train_images[], matrix *train_labels[], int train_size, int epochs, int mini_batch_size, double eta,
        matrix *test_images[], matrix *test_labels[], int test_size) {

    matrix **mini_batch_images = (matrix**)malloc(mini_batch_size * sizeof(char*));
    matrix **mini_batch_labels = (matrix**)malloc(mini_batch_size * sizeof(char*));
    int i,k;
    int *array = (int*)malloc(train_size * sizeof(int));
    for (i = 1; i <= epochs; i++) {
        //混洗，即改变原来train_images的数组指针的指向
        randomShuffle(array, train_size);
        //printf("SGD: testtttttt1\n");
        /*
        for (k = 0; k < train_size; k++) {
            train_images[k] = train_images[array[k]];
            train_labels[k] = train_labels[array[k]];
        }
        */
        //以上是错误的，交换过程中会使得元素相互覆盖了，其实可以不改变原来的顺序，
        //我们只是在访问train_images的时候按照混洗顺序访问即可
        for (int start = 0; start < train_size; start+=mini_batch_size) {
            for(k = 0; k < mini_batch_size; k++) {
                mini_batch_images[k] = train_images[array[start+k]];
                mini_batch_labels[k] = train_labels[array[start+k]];
            }
            //printf("SGD: testtttttt2\n");
            update_mini_batch(mini_batch_images, mini_batch_labels, mini_batch_size, eta);
            //printf("SGD: testtttttt3\n");
        }
        if (test_images != NULL && test_labels != NULL) {
            printf("Epoch %d: %d / %d\n", i, evaluate(test_images, test_labels, test_size), test_size);
            /*
            for (k = 0; k < num_layers-1; k++) {
                printf("weights %d<>%d:\n", k+1, k+2);
                printMatrix(weights[k]);
            }
            */
        }
        else {
            printf("Epoch %d complete\n", i);
        }
    }


    free(array);
    free(mini_batch_images);
    free(mini_batch_labels);

    return 0;
}
```

C语言实现版本

FOLDERS
- demo
  - mnist-parser
  - matrix.c
  - matrix.h
  - mnist_reader.c
  - mnist_reader.h
  - nn
  - nn.c
  - nn1.c
  - test_images.txt
  - test_labels.txt
  - test_matrix
  - test_matrix.c
  - test_reader
  - test_reader.c
  - train_images.txt
  - train_labels.txt

nn.c

```c

int update_mini_batch(matrix *mini_batch_images[], matrix *mini_batch_labels[], int mini_batch_size, double eta) {
    int i;
    // 初始化矩阵数组，用于存放累deltaC的累加值，因为是batch
    matrix **nabla_w = (matrix**)malloc((num_layers-1) * sizeof(char*));
    matrix **nabla_b = (matrix**)malloc((num_layers-1) * sizeof(char*));
    for (i = 0; i < num_layers-1; i++) {
        nabla_w[i] = newMatrix(weights[i]->rows, weights[i]->cols);
        nabla_b[i] = newMatrix(biases[i]->rows, biases[i]->cols);
    }
    //printf("update_mini_batch: testttttt1\n");
    //minibatch sum
    matrix **delta_nabla_w = (matrix**)malloc((num_layers-1) * sizeof(char*));
    matrix **delta_nabla_b = (matrix**)malloc((num_layers-1) * sizeof(char*));
    for (i = 0; i < mini_batch_size; i++) {
        //注意！！不需要对delta_nabla_w[i]和delta_nabla_b[i]初始化空间，因为backprop内部会分配空间给他们；
        /*
        for (int j = 0; j < num_layers-1; j++) {
            delta_nabla_w[j] = newMatrix(weights[j]->rows, weights[j]->cols);
            delta_nabla_b[j] = newMatrix(biases[j]->rows, biases[j]->cols);
        }
        */

        //printf("update_mini_batch: testttttt2\n");
        // 对一个x，y反向传播一次
        backprop(mini_batch_images[i], mini_batch_labels[i], delta_nabla_w, delta_nabla_b);

        //printf("update_mini_batch: testttttt3\n");
        for (int j = 0; j < num_layers-1; j++) {
            sum(nabla_w[j], delta_nabla_w[j], nabla_w[j]);
            sum(nabla_b[j], delta_nabla_b[j], nabla_b[j]);
        }

        // 运行完一次backprop，delta_nabla_w 和delta_nabla_b便不再使用，及时释放

        for(int j = 0; j < num_layers-1; j++){
            deleteMatrix(delta_nabla_w[j]);
            deleteMatrix(delta_nabla_b[j]);
        }

    }

    //释放delta_nabla_w和delta_nabla_b指针数组自己
    free(delta_nabla_b);
    free(delta_nabla_w);

    // 更新w和b
    for (i = 0; i < num_layers-1; i++) {
        multiplyMatrix(nabla_w[i], eta/mini_batch_size);
        minus(weights[i], nabla_w[i], weights[i]);

        multiplyMatrix(nabla_b[i], eta/mini_batch_size);
        minus(biases[i], nabla_b[i], biases[i]);
    }

    //释放空间
    for (i = 0; i < num_layers-1; i++) {
        deleteMatrix(nabla_w[i]);
        deleteMatrix(nabla_b[i]);
    }
    free(nabla_w);
    free(nabla_b);

}
```

C语言实现版本

FOLDERS
- demo
  - mnist-parser
  - matrix.c
  - matrix.h
  - mnist_reader.c
  - mnist_reader.h
  - nn
  - nn.c
  - nn1.c
  - test_images.txt
  - test_labels.txt
  - test_matrix
  - test_matrix.c
  - test_reader
  - test_reader.c
  - train_images.txt
  - train_labels.txt

nn.c

```c
int backprop(matrix *x, matrix *y, matrix *nabla_w[], matrix *nabla_b[]) {
    int i;

    //feedforward
    matrix *activation = copyMatrix(x);
    matrix **activations = (matrix**)malloc(num_layers * sizeof(char*));//存储所有的激活层
    matrix **zs = (matrix**)malloc(num_layers * sizeof(char*));          //存储所有的加权层
    activations[0] = activation;
    //printf("backprop: testttttttt0\n");
    for (i = 0; i < num_layers-1; i++) {
        matrix *z = newMatrix(weights[i]->rows, activation->cols);
        product(weights[i], activation, z); // z = W*a
        sum(z, biases[i], z);               // z = W*a + b
        zs[i+1] = copyMatrix(z);            // zs.add(z)zs[1]zs[2]..zs[num_layers-1]对应2,3...L层,zs[0]不用, 因为没有
        funcMatrix(z, sigmoid);             //
        activation = z;                     // a = sigmoid(z)
        activations[i+1] = activation;      // as.add(a)as[0]as[1]..as[num_layers-1]对应1,2...L层
    }
    //printf("backprop: testttttttt1\n");
    //backward pass
    //首先可以直接根据y算出最后一层的delta
    matrix *nabla_c = cost_derivative(activations[num_layers-1], y);
    matrix *nabla_z = copyMatrix(zs[num_layers-1]);
    funcMatrix(nabla_z, sigmoid_prime);
    matrix *delta = newMatrix(nabla_z->rows, nabla_z->cols);
    scalarProduct(nabla_c, nabla_z, delta);  // deltaL = nabla_C (*) sigmoid_prime(z)
    deleteMatrix(nabla_c);
    deleteMatrix(nabla_z);
    //printf("backprop: testttttttt2\n");
    nabla_b[num_layers-2] = delta;
    transposeSelf(activations[num_layers-2]); // 倒数第二层的激活的转置 a(L-1).T
    //printf("backprop: testttttttt3\n");//num_layers-2才是L-1<>L之间的权值矩阵，即最后一个权值矩阵
    matrix *dp = newMatrix(weights[num_layers-2]->rows, weights[num_layers-2]->cols);
    //printf("backprop: testttttttt4\n");
    product(delta, activations[num_layers-2], dp); // delta product a(L-1).T
    nabla_w[num_layers-2] = dp;

    //printf("backprop: testttttttt5\n");

    int l;
    // 从倒数第二层开始反向传播计算每一层的delta，并保存每一层的偏导nabla_b 和nabla_w
    // 注意！
    // 对于zs[0,1,2...num_layers-1]分别对应1,2,3...L层的z，但是第一层没有z，所以不用
    // 对于as[0,1,2...num_layers-1]分别对应1,2,3...L层的a，没一层都有，第一层就是输入自己
    // 对于weights[0,1,2...num_layers-2]分别对应1<>2, 2<>3... L-1<>L之间的权值矩阵
    // 对于 biases[0,1,2...num_layers-2]分别对应1<>2, 2<>3... L-1<>L之间的偏差矩阵
    // 对于层数，是按照下标从0到num_layers-1的，分别对应1,2,3...L，这里的l从L-1开始递减到1
    for (l = num_layers-2; l >= 1; l--) {
        matrix *sp = copyMatrix(zs[l]);
        funcMatrix(sp, sigmoid_prime);
        matrix *wt = copyMatrix(weights[l]);
        //printf("backprop: testttttttt6\n");
        transposeSelf(wt);
        matrix *deltal = newMatrix(biases[l-1]->rows, biases[l-1]->cols);
        int ret = product(wt, nabla_b[l], deltal);
        if (ret == -2) {
            printf("wrong size: wt size: (%d, %d) nabla_b[%d] size: (%d, %d), deltal size: (%d, %d)\n",
                wt->rows, wt->cols, l, nabla_b[l]->rows, nabla_b[l]->cols, deltal->rows, deltal->cols);
        }
        scalarProduct(deltal, sp, deltal);  // delta[l] = (w[l+1].T * delta[l+1]) (*) sp(zl)
        //printf("backprop: testttttttt7\n");
        nabla_b[l-1] = deltal;
        //printf("nabla_b[%d]\n", l-1);
        //printMatrix(nabla_b[l-1]);
        //printf("backprop: testttttttt8\n");
        //printMatrix(activations[l-1]);
```

附录

- 代码网址
  - https://github.com/kitianFresh/neural-networks-by-c
- 使用工具
  - Valgrind　C语言程序内存泄漏检测工具
- 算法
  - O(1)空间转置矩阵
  - O(n)时间混洗数组
  - 高斯分布随机数
- 参考
- http://neuralnetworksanddeeplearning.com/

Thank you