神经网络的编程基础

- 本节以二分类问题为例，即输出仅包含两种情况的分类问题

符号定义

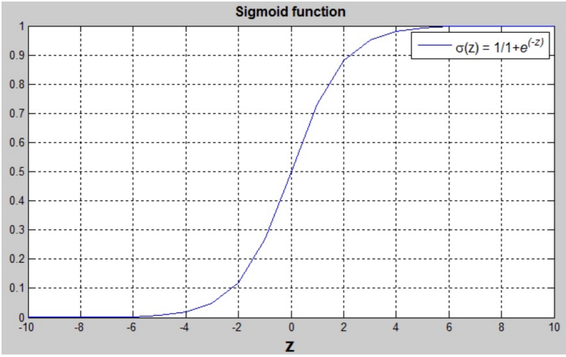
- x：表示输入，维度为 $(n_x, 1)$
- y：表示输出，取值为 $(0, 1)$
- $(x^{(i)}, y^{(i)})$：表示第 i 组数据
- m：表示训练集的样本个数
- m_{test}：表示测试集的样本个数
- $X = [x^{(1)}, x^{(2)}, \ldots, x^{(m)}]$：表示所有训练数据集的输入值，维度为 (n_x, m)
- $Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$：表示所有训练数据集的输出值，维度为 $(1, m)$

逻辑回归

- 逻辑回归适用于二分类问题。其公式为：

$$\hat{y} = P(y = 1 \mid x), where\ 0 \leq \hat{y} \leq 1$$

  - 具体的参数包括：
    - 输入特征向量：$x \in \mathbb{R}^{n_x}$
    - 训练标签：$y \in \{0, 1\}$
    - 权重：$w \in \mathbb{R}^{n_x}$
    - 输出：$\hat{y} = \sigma(w^T x + b)$
    - sigmoid函数：$s = \sigma(w^T x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$



逻辑回归的代价函数

- 为了训练参数 w 和 b，我们需要定义一个代价函数
    - 最小二乘函数会导致局部最优（非凸优化），不适用于逻辑回归
    - 这里选择如下的损失函数（交叉熵函数）：

$$L(\hat{y^{(i)}}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$

        - 该损失函数本质上是由极大似然估计得出的
    - 因此代价函数为：

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

        - 损失函数计算单个训练样本的误差，而代价函数是整个训练集损失函数的平均

梯度下降

- 我们需要找出最小化代价函数的 w 和 b
    - 易证明代价函数是凸函数（能够找出全局最优）
- 这里采用梯度下降法，不断沿着梯度方向更新参数，直至收敛

$$w = w - \alpha * \frac{d(J(w, b))}{dw}$$
$$b = b - \alpha * \frac{d(J(w, b))}{db}$$

    - α 称为学习速率，控制梯度更新的步幅
    - 在逻辑回归中一般初始化参数为 0

逻辑回归中的梯度下降

- 在逻辑回归中，梯度下降涉及到复合求导，需要基于链式法则求解
    - 课程中介绍了计算图的方法，能够更加直观地求解复合导数
    - 这其实可以看做一种简单的反向传播
- 下面给出求解含有 m 个样本的逻辑回归的梯度下降的伪代码
    - 变量名如下：

```
X1                    Feature
X2                    Feature
W1                    Weight of the first feature.
W2                    Weight of the second feature.
B                     Logistic Regression parameter.
M                     Number of training examples
Y(i)                  Expected output of i
```

- 基于复合求导得出的导数如下：

```
d(a)  = d(l)/d(a) = -(y/a) + ((1-y)/(1-a))
d(z)  = d(l)/d(z) = a - y
d(W1) = X1 * d(z)
d(W2) = X2 * d(z)
d(B)  = d(z)
```

- 伪代码如下：

```
J = 0; dW1 = 0; dW2 =0; dB = 0;                 # Devs
W1 = 0; W2 = 0; B=0;                            # Weights
for i = 1 to m
    # Forward pass
    z(i) = W1*X1(i) + W2*X2(i) + b
    a(i) = Sigmoid(z(i))
    J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

    # Backward pass
    dz(i) = a(i) - Y(i)
    dW1 += dz(i) * X1(i)
    dW2 += dz(i) * X2(i)
    dB  += dz(i)
J /= m
dW1/= m
dW2/= m
dB/= m

# Gradient descent
W1 = W1 - alpha * dW1
W2 = W2 - alpha * dW2
B = B - alpha * dB
```

  - 上述伪代码实际上存在两组循环（迭代循环没有写出），会影响计算的效率
  - 我们可以使用**向量化**来减少循环

向量化

- 向量化可以避免循环，减少运算时间
- Numpy 的函数库基本都是向量化版本

- 向量化可以在 CPU 或 GPU 上实现（通过 SIMD 操作），GPU 上速度更快

向量化逻辑回归

- 下面将仅使用一组循环来实现逻辑回归
    - 输入变量为：

    ```
    X        Input Feature, X shape is [Nx,m]
    Y        Expect Output, Y shape is [Ny,m]
    W        Weight, W shape is [Nx,1]
    b        Parameter, b shape is [1,1]
    ```

    - 向量化后的伪代码如下：

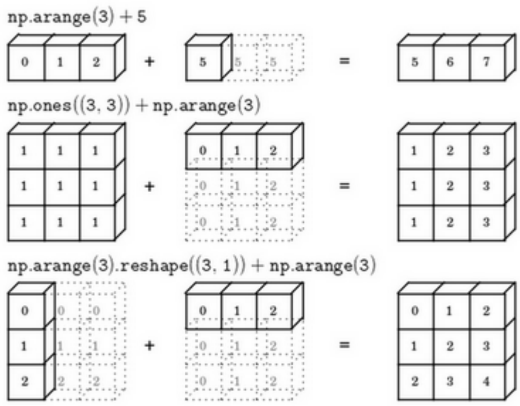
    ```python
    W = np.zeros((Nx, 1))
    b = 0
    dW = np.zeros((Nx, 1))
    db = 0

    for iter in range(1000):
        Z = np.dot(W.T, X) + b       # Vectorization, then broadcasting, Z shape is (1, m)
        A = 1 / (1 + np.exp(-Z))     # Vectorization, A shape is (1, m)
        dZ = A - Y                   # Vectorization, dZ shape is (1, m)
        dW = np.dot(X, dZ.T) / m     # Vectorization, dW shape is (Nx, 1)
        db = np.sum(dZ) / m          # Vectorization, db shape is (1, 1)

        W = W - alpha * dW
        b = b - alpha * db
    ```

Python/Numpy 使用笔记

- 在 Numpy 中，`obj.sum(axis = 0)` 按列求和，`obj.sum(axis = 1)` 按行求和，默认将所有元素求和
- 在 Numpy 中，`obj.reshape(1, 4)` 将通过广播机制（broadcasting）重组矩阵
    - reshape 操作的调用代价极低，可以放在任何位置
    - 广播机制的原理参考下图：



- 关于矩阵 shape 的问题：
    - 如果不指定一个矩阵的 shape，将生成 "rank 1 array"，会导致其 shape 为 `(m, )`，无法进行转置
        - 对于这种情况，需要进行 reshape
    - 可以使用 `assert(a.shape == (5,1))` 来判断矩阵的 shape 是否正确
- 计算 Sigmoid 函数的导数：

```
s = sigmoid(x)
ds = s * (1 - s)
```

- 如何将三维图片重组为一个向量：

```
v = image.reshape(image.shape[0]*image.shape[1]*image.shape[2],1)
```

- 归一化输入矩阵后，梯度下降将收敛得更快

构建神经网络

- 构建一个神经网络一般包含以下步骤
    1. 定义神经网络的结构
    2. 初始化模型参数
    3. 重复以下循环直至收敛：
        - 计算当前的代价函数（前向传播）
        - 计算当前的梯度（反向传播）
        - 更新参数（梯度下降）
- 数据集的预处理与超参数（如学习速率）的调整十分重要