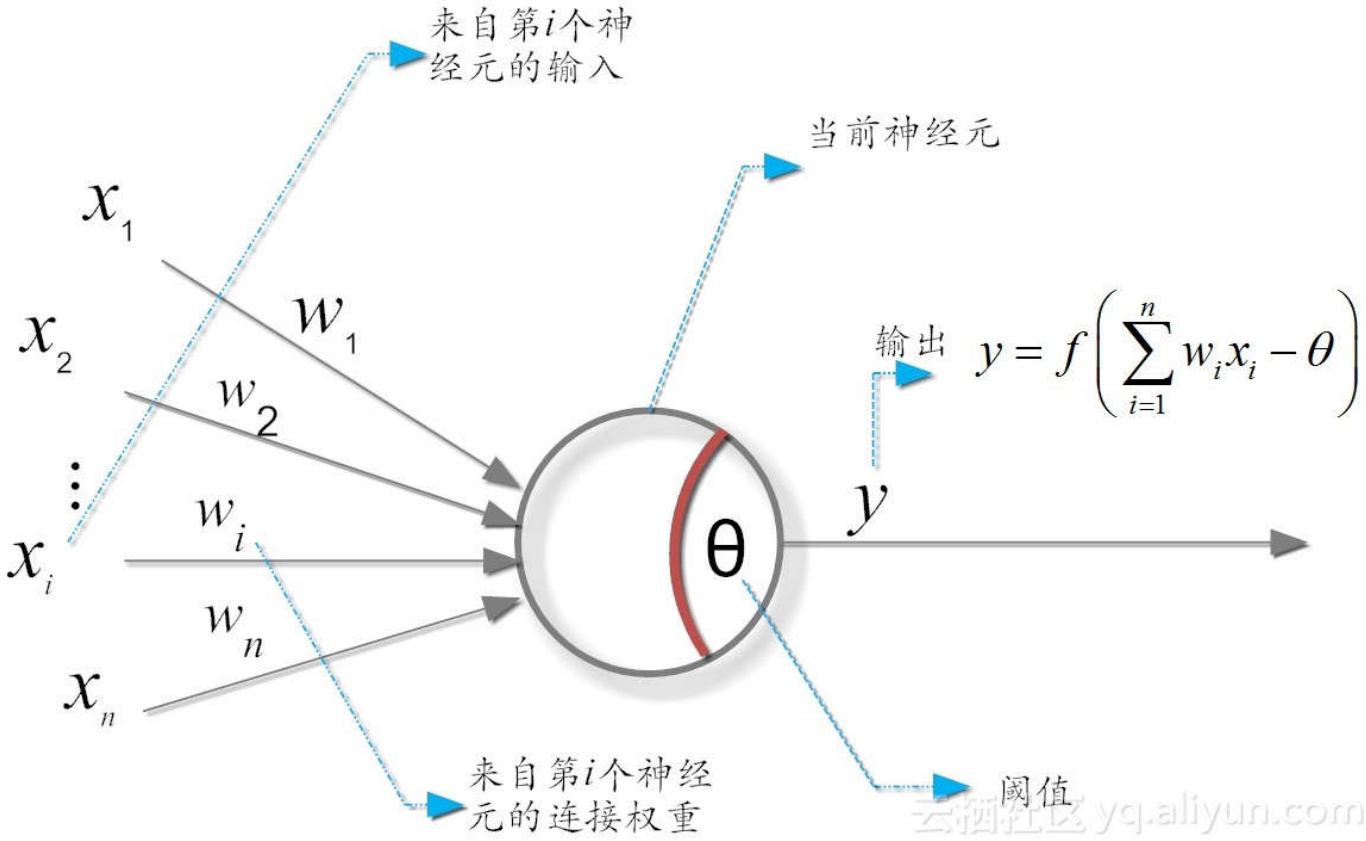
神经网络

1. 神经元模型

- **M-P神经元模型**

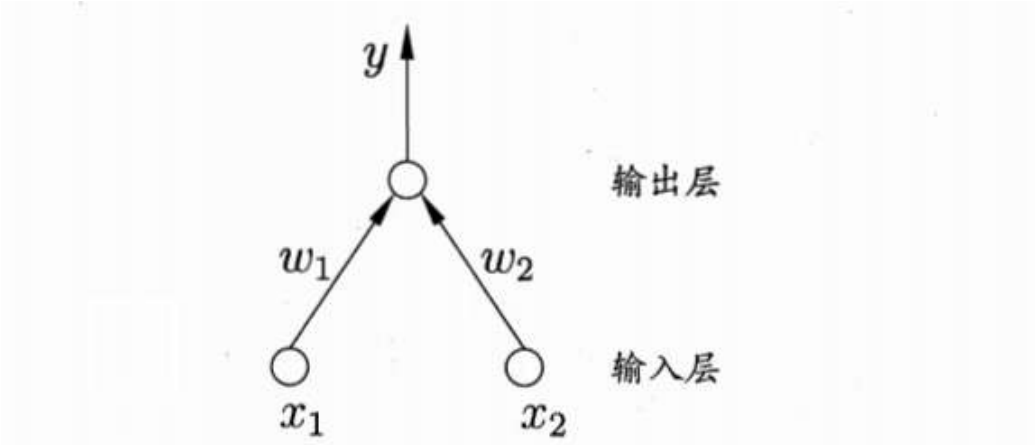


几个概念:
阈值θ :当输入的加权和>阈值时, 神经元处于激活状态, 否则的话, 处于未激活状态
激活函数 f:处理加权和和阈值产生神经元输出
激活函数的选择:
最理想: 阶跃函数$\text{sgn}(x) = \begin{cases} 1, x > 0 \\ 0, x < 0 \end{cases}$
性质最好: 挤压函数$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$
连接上面的神经元就得到最简单的神经网络了.

2. 感知机和多层网络

2.1 感知机

- **感知机**:又称阈值逻辑单元，是由输入层和MP输出层构成的简单神经网络



两个输入神经元的感知机网络结构示意图

特点:

- 输出层是MP神经元
- 容易调整w, θ和通过阶跃函数实现逻辑运算(与或非)

数学模型:

为了方便研究, 将阈值θ视作一个固定输入为-1的哑结点的参数记为w_{n+1}
学习的过程就是调整参数的过程
感知机的权重调整过程:

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

$\eta \in (0, 1)$是学习率

2.2 多层神经网络

为什么引入多层神经网络?

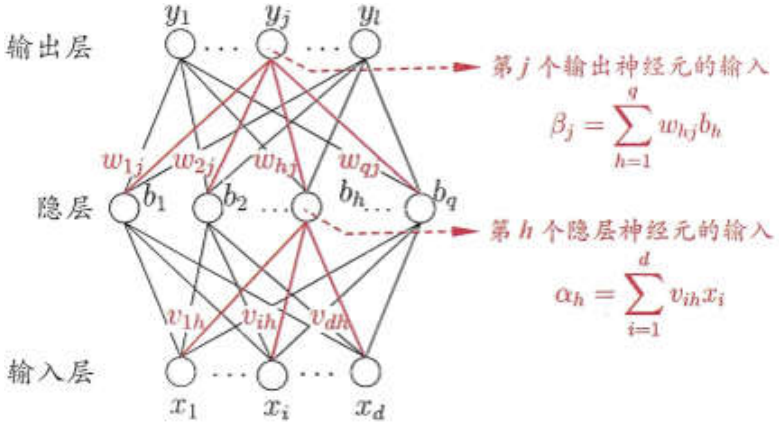
使用感知机得到的只是线性模型, 不能处理线性不可分问题. 为了处理非线性可分问题, 引入多层神经网络.

几个概念:

- 隐藏层:在输入层和输出层之间的层, 都是拥有激活函数的功能神经元.
- 前馈:网络在拓扑结构上没有环或者回路.
- 全连接: 每层神经元与下一层神经元全连接.

3. 反向传播算法(误差逆传播)

- **BP算法**(Error BackPropagation):是一种学习算法, 意在于习得神经网络的参数.
- **模型描述**: 对于给定数据集$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}, x_i \in \mathbb{R}^d, y_i \in \mathbb{R}^l$, 即输入向量$x$应是$d$个属性, 输出向量$y$应是$l$维, 以拥有$d$个输入神经元, l个输出神经元, q个隐层神经元的神经网络为例:输出层的第j个神经元的阈值用θ_j表示, 隐层第h个神经元阈值用γ_h表示, 输入层第i个神经元与隐层的h个神经元之间的连接权为v_{ih}, 隐层第h个神经元与输出层第j个神经元之间的连接权为w_{hj}. 所以隐层第h个神经元接受的输入为:$\alpha_h = \sum_{i=1}^{d} v_{ih}x_i$, 输出层第$j$个神经元接收到的输入为$\beta_j = \sum_{h=1}^{q} w_{hj}b_n$
  图示如下:



考虑训练例(x_k, y_k), 假定神经网络的输出是$\hat{y}_k = (\hat{y}_1^k, \hat{y}_1^k, \cdots, \hat{y}_1^k)$, 即：$\hat{y}_j^k = f(\beta_j - \theta_j)$
网络产生的均方误差为:

$$E_k = \frac{1}{2}\sum_{j=1}^{l}(\hat{y}_j^k - y_j^k)^2$$

为了确定参数, 使用BP算法进行迭代学习, 基于梯度下降策略.

- **梯度下降算法:**
  梯度下降策略每次迭代的规则如下:

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}}$$

由链式法则:

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k}\frac{\partial \hat{y}_j^k}{\partial \beta_j}\frac{\partial \beta_j}{\partial w_{hj}}$$

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h$$

$$\frac{\partial E_k}{\partial \hat{y}_j^k} = \hat{y}_j^k - y_j^k$$

$$\frac{\partial \hat{y}_j^k}{\partial \beta_j} = \frac{\partial f(\beta_j - \theta_j)}{\partial \beta_j} = \frac{\partial f(\beta_j - \theta_j)}{\partial (\beta_j - \theta)}$$

sigmoid函数具有以下性质:

$$f^{'}(x) = f(x)(1 - f(x))$$

所以:
记

$$g_j = -\frac{\partial E_k}{\partial \hat{y}_j^k}\frac{\partial \hat{y}_j^k}{\partial \beta_j} = (y_j^k - \hat{y}_j^k) f^{'}(\beta_j - \theta_j) = (y_j^k - \hat{y}_j^k)\hat{y}_j^k(1 - \hat{y}_j^k)$$

所以,

$$\Delta w_{hj} = \eta g_j b_h$$

同理,

$$\Delta\theta_j = -\eta g_j$$
$$\Delta v_{ih} = \eta e_h x_j$$
$$\Delta\gamma_h = -\eta e_h$$

其中:

$$e_h = -\frac{\partial E_k}{\partial b_h}\frac{\partial b_h}{\partial \alpha_h} = -\sum_{j=1}^{l}\frac{\partial E_k}{\partial \beta_j}\frac{\partial \beta_j}{\partial b_h}\frac{\partial b_h}{\partial \alpha_h} = \sum_{j=1}^{l} g_j w_{hj} f^{'}(\alpha_h - \gamma_h) = \sum_{j=1}^{l} g_j w_{hj} b_h(1-b_h)$$

学习率η的含义: 学习率介于0,1之间, 控制每次调整的步长, 过大容易震荡, 过小容易速度过慢.层与层之间的学习率不必一致, 但是每一层的阈值和权系数的学习率必须相同.

算法步骤:

1. 神经网络前向传播, 计算输出$\hat{y}_k$
2. 反向传播计算Δ, 调整网络参数
3. 判断是否达到停止条件, 若是结束, 否则goto1

- ***标准BP算法与累积BP算法***
  注意到上述BP算法是针对数据集中的一个数据进行的优化, 因此需要更新的次数很多, 而累积BP算法是针对整个数据集的优化, 每次调整参数时遍历整个数据集, 求得累积误差, 因此参数更新频率很低. 常用的方法是使用累积BP算法进行充分调整后, 使用标准BP算法加速.
- ***过拟合问题***
  由于网络对于数据集特征的表示过好, 造成过拟合
  解决办法:
  - 早停(Early Stopping): 将数据集拆成验证集和训练集两个部分, 一旦训练集误差降低, 验证集误差升高, 就停止训练.
  - 正则化(Regularization): 综合考虑网络复杂度和误差, 比如可以:

    $$E = \lambda\frac{1}{m}\sum_{k=1}^{m}E_k + (1-\lambda)\sum_i w_i^2$$

    在上式中第二项用阈值权重平方和描述网络复杂度, 倾向于生成系数小的光滑网络, 防止过拟合.

4. 局部最小与全局最小

- ***概念:*** 局部最小指的是在其邻域内, 有$E(w,\theta) \geq E(w^*,\theta^*)$, 全局最小指的是在全空间内有$E(w,\theta) \geq E(w^*,\theta^*)$
  可能有很多个局部最小, 但是只有一个全局最小. 我们希望参数调优的过程中找到全局最小. 按照梯度下降的算法我们很容易在有多个局部最小的空间中陷入局部最小.
- ***跳出局部最小的方法:***
  1. 以多组不同的参数初始化神经网络, 是之陷入多个局部最小, 找到误差最小的一个就可以认为是全局最小
  2. 模拟退火技术: 每步迭代过程中以一定概率接受比当前差的结果, 以跳出局部最小, 但是随着迭代过程的进行这一概率应该逐渐降低, 以保证收敛.
  3. 随机梯度下降: 计算梯度时加入随机扰动, 即使陷入局部最小, "梯度"仍可能不为0, 从而跳出.
  4. 遗传算法

5. 其他神经网络

- RBF神经网络
  通常上是二层网络, 具有一个隐层, 以径向基函数作为激活函数, 输出为隐层输出的线性组合.
- ART神经网络
  竞争型学习是常用的无监督学习方法, 网络的输出神经元互相竞争, 每个时刻只有一个神经元获胜被激活, 其他神经元处于抑制状态, 这种机制被称为赢者通吃原则.
- SOM神经网络
  是一种竞争学习型无监督神经网络, 将高维数据映射到低维, 保持高维数据的拓扑结构, 即将高维数据相似的样本点, 映射到网络临近的神经元, 输出层神经元通常以矩阵形式排布在二维空间, 训练的目标是找到合适的权向量使得其维持拓扑结构.
- 级联相关网络
  结构自适应神经网络的重要代表, 训练的目标是找到合适的网络结构.
  有两个主要成分:级联和相关
  级联是指加入新的隐藏层神经元, 但是维持输入权值不变, 相关是指通过最大化新神经元输出和网络误差的相关性；来训练参数.
  特点: 训练速度快, 但是容易陷入过拟合.
- Elman网络
  递归神经网络的代表, 隐层神经元的输出被反馈回来, 与下一时刻的输入信号一起作为网络的输入.
- Boltzmann机
  一种基于网络能量的模型, 神经元都是布尔类型的. 1表示激活, 0表示未激活. 以向量s表示n个神经元的状态, w_ij表示神经元ij之间的连接权, \theta_i代表神经元i的阈值. 定义Boltzmann机能量为:

$$E(s) = -\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij}s_i s_j - \sum_{i=1}^{n}\theta_i s_i$$

  如果更新过程不依赖输入值顺序, 最后会达到玻尔兹曼分布.
  状态向量出现的概率仅由其能量和所有可能出现的向量的能量确定:

$$P(s) = \frac{\exp\{-E(s)\}}{\sum_t \exp\{-E(t)\}}$$

  训练原理:
  将每一个样本视为一个状态向量, 是之出现的概率尽可能大.

6. 深度学习

复杂网络的代表
特点是具有很深层的神经网络, 难以使用BP算法进行训练, 因为反向传播时很容易不收敛.
训练方法:

1. 无监督逐层训练:预训练+微调
   预训练: 逐层训练隐层, 将训练后的输出作为下一层的输入.
   微调: 可以使用BP算法
2. 权共享: 让一组神经元使用相同的连接权
   减少了需要训练的参数数目, 可以使用BP算法, 典型代表是卷积神经网络(CNN)

7. 神经网络的优化方法

1. 指数衰减学习率
   学习率计算公式如下:

$$Learning\_rate = 初始值 \times 衰减率 \times \frac{当前训练轮数}{每轮训练数据数}$$

2. 滑动平均:
   定义滑动平均(影子), 来描述参数在过往一段时间的平均值以增加泛化性.

$$当前影子 = 衰减率 \times 上一步影子 + (1 - 衰减率) \times 当前参数$$

$$衰减率 = \min(滑动平均衰减率, \frac{1 + 轮数}{10 + 轮数})$$

   tensorflow代码:

```
ema = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DACAY, global_step)
ema_op = ema.apply(tf.trainable_varibles())
with tf.control_dependencies([train_step, ema_op]):
   train_op = tf.no_op(name='train')
para_average = ema.average()
```

3. 正则化方法

$$Loss = loss1(y - y\_) + Para \times loss2(w)$$

   第一项描述数据误差, 第二项描述模型复杂度, 防止过拟合
   loss2描述网络复杂度有两种, 有L1和L2

$$loss2_{L1} = \sum_i |w_i|$$
$$loss2_{L1} = \sum_i |w_i^2|$$

   tenserflow函数

```
loss(w) = tf.contrib.layers.l1_regularize(regularize)(w)
tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularize)(w))
loss = cem + tf.add_n(tf.get_collection('loss'))
```

8. 神经网络的通用搭建模式

前向传播:

```
# 定义前向传播过程
def forward(x, regularizer):
    w=
    b=
    y=
    return y

# 获取权重w
def get_weight(shape, regularizer):
    w=tf.Variable() #赋初值
    tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularizer)(w))
    return w

# 获取偏置项b
def get_bias(shape):
    b = tf.Variable() #赋初值
    return b
```

反向传播

```
def backword():
    x = tf.placeholder()#占位
    y_ = tf.placeholder()#占位
    y = forward.forward(x, REGULARIZER)#正向传播过程
    global_step = tf.Variable(0, trainable = False)#训练步数
    loss = f(y-y_)( + tf.add_n(tf.get_collection('losses')))#可选正则化项的损失函数
    """
    f(y-y_) = tf.reduce_mean(tf.square(y-y_))#均方误差
     tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y, lables=tf.argmax(y_1,))
    f(y-y_) = tf.reduce_mean(ce)#交叉熵
    """
    """
    #指数衰减学习率
    global_step = tf.Variable(0, trainable=False)
    learning_rate = tf.train.exponential_decay(LEARNING_RATE_BASE, global_step, LEARNING_RATE_STEP, LEARN
    #learning_rate_step表示多少轮训练之后更新学习率
    #一般取learning_rate_step = 训练数据总数/每轮数据数，因为训练数据的更新步长为1，经过rate_step回到最初的起点.
    """
    """
    #滑动平均
    ema = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step)
    ema_op = ema.apply(tf.trainable_variables())
    with tf.control_dependencies([train_step, ema_op]):
        train_op = tf.no_op(name='train')
    """
    train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss) #指出训练方法和优化函数
    with tf.Session() as sess:
        init_op = tf.global_variables_initializer()
        sess.run(init_op)#初始化
        for i in range(STEPS):
            sess.run(train_step, feed_dict = {x: X[start: end], y_:Y_[start: end]})
            if i % step == 0:
                loss_v.run(loss_total, feed_dict = {x:X, y_:Y_})
                print
    if __name__ == '__main__':
        backward()
```