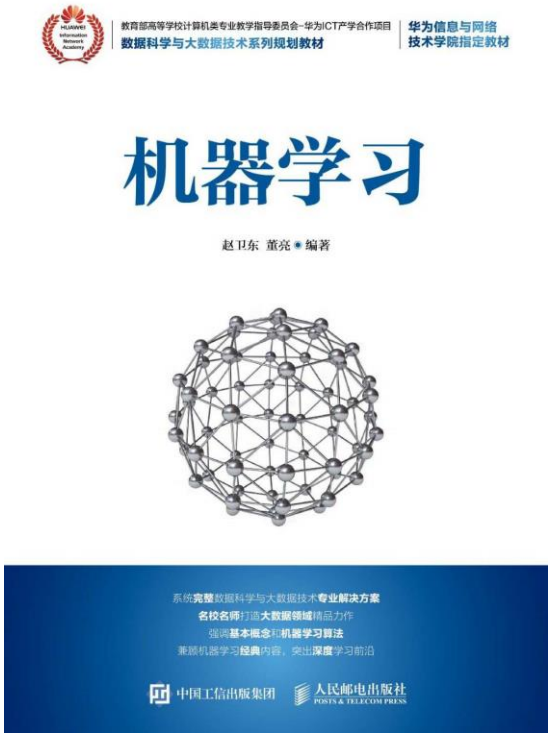


机器学习
第6章　神经网络

复旦大学 **赵卫东** 博士

wdzhao@fudan.edu.cn

章节介绍

- 人工神经网络(Artificial Neural Netork,即ANN)是由简单神经元经过相互连接形成网状结构，通过调节各连接的权重值改变连接的强度，进而实现感知判断
- 反向传播(Back Propagation, BP) 算法的提出进一步推动了神经网络的发展。目前，神经网络作为一种重要的数据挖掘方法，已在医学诊断、信用卡欺诈识别、手写数字识别以及发动机的故障诊断等领域得到了广泛的应用
- 本章将介绍神经网络基本分类，包括前馈神经网络、反馈神经网络、自组织神经网络等常用的神经网络模型。重点介绍神经网络的概念和基本原理，为后续深度学习章节的学习打下基础

章节结构

- 神经网络介绍
  - 前馈神经网络
  - 反馈神经网络
  - 自组织神经网络
- 神经网络相关概念
  - 激活函数
  - 损失函数
  - 学习率
  - 过拟合
  - 模型训练中的问题
  - 神经网络效果评价
- 神经网络的应用

神经网络介绍

- 传统神经网络结构比较简单，训练时随机初始化输入参数，并开启循环计算输出结果，与实际结果进行比较从而得到损失函数，并更新变量使损失函数结果值极小，当达到误差阈值时即可停止循环
- 神经网络的训练目的是希望能够学习到一个模型，实现输出一个期望的目标值。学习的方式是在外界输入样本的刺激下不断改变网络的连接权值。传统神经网络主要分为一下几类：前馈型神经网络，反馈型神经网络和自组织神经网络。这几类网络具有不同的学习训练算法，可以归结为监督型学习算法和非监督型学习算法

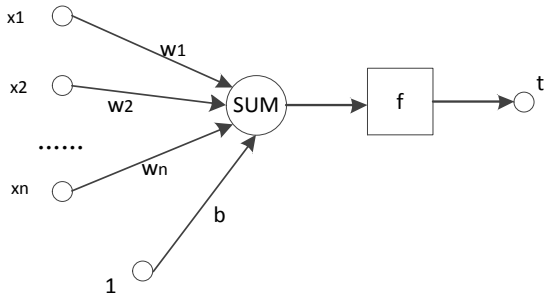
前馈神经网络

- 前馈神经网络(Feed Forward Neural Network) 是一种单向多层的网络结构，即信息是从输入层开始，逐层向一个方向传递，一直到输出层结束。所谓的“前馈”是指输入信号的传播方向为前向，在此过程中并不调整各层的权值参数，而反传播时是将误差逐层向后传递，从而实现使用权值参数对特征的记忆，即通过反向传播(BP) 算法来计算各层网络中神经元之间边的权重。BP算法具有非线性映射能力，理论上可逼近任意连续函，从而实现对模型的学习

感知器

- 感知器是一种结构最简单的前馈神经网络，也称为感知机，它主要用于求解分类问题
- 一个感知器可以接收n个输入$x=(x_1, x_2, \ldots, x_n)$，对应$n$个权值$w=(w_1, w_2, \ldots, w_n)$，此外还有一个偏置项阈值，就是图中的$b$，神经元将所有输入参数与对应权值进行加权求和，得到的结果经过激活函数变换后输出，计算公式如下：

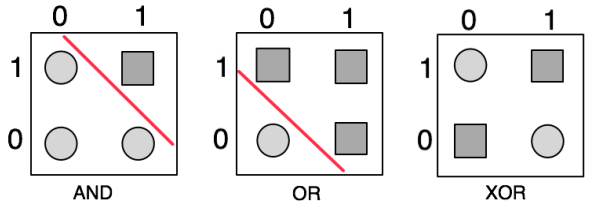
$$y = f(x * w + b)$$



感知器

- 神经元的作用可以理解为对输入空间进行直线划分，单层感知机无法解决最简单的非线性可分问题----异或问题
- 感知器可以顺利求解与(AND) 和或(OR)问题，但是对于异或(XOR)问题，单层感知机无法通过一条线进行分割

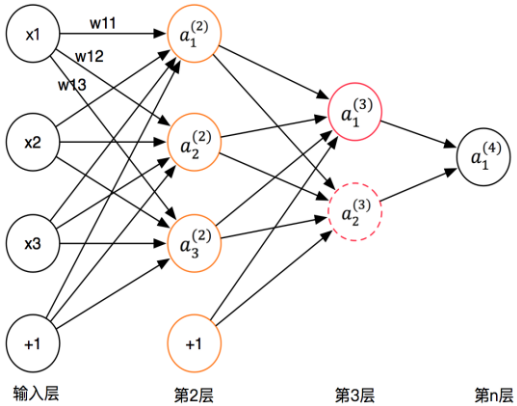
输入		输出
a	b	y
0	1	1
0	0	0
1	1	0
1	0	1



BP神经网络

- BP (Back Propagation)神经网络也是前馈神经网络，只是它的参数权重值是由反向传播学习算法进行调整的
- BP 神经网络模型拓扑结构包括输入层、隐层和输出层，利用激活函数来实现从输入到输出的任意非线性映射，从而模拟各层神经元之间的交互
- 激活函数须满足处处可导的条件。例如，Sigmoid函数连续可微，求导合适，单调递增，输出值是0~1之间的连续量，这些特点使其适合作为神经网络的激活函数

BP神经网络



- 图中网络结构对应的计算公式如下:

$$a_1^{(2)} = f(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_{11}^{(1)})$$

$$a_2^{(2)} = f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_{12}^{(1)})$$

$$a_3^{(2)} = f(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_{13}^{(1)})$$

$$h_{W,b}(x) = \mathrm{a}_1^{(3)} = f(w_{11}^{(2)}a_1 + w_{12}^{(2)}a_2 + w_{13}^{(2)}a_3 + b_{21}^{(2)})$$

BP神经网络

- 式中的w_{ij}就是相邻两层神经元之间的权值，它们是在训练过程中需要学习的参数，$a_1^{(2)}$中表示第2层的第1个神经元，公式中的（6.1）、（6.2）、（6.3）分别求出了3个神经元的输出结果，而$h_{W,b}(x)$表示第3层第1个神经元$a_1^{(3)}$的值。图中箭头所指的方向为前向传播的过程，即所有输入参数经过加权求和之后，将结果值依次向下一层传递，直到最后输出层，层数越多、层中神经元越多，形成的权重值参数就越多

BP神经网络

- BP神经网络训练过程的基本步骤可以归纳如下
  - 初始化网络权值和神经元的阈值，一般通过随机的方式进行初始化
  - 前向传播:计算隐层神经元和输出层神经元的输出
  - 后向传播:根据目标函数公式修正权值w_{ij}
- 上述过程反复迭代，通过损失函数和成本函数对前向传播结果进行判定，并通过后向传播过程对权重参数进行修正，起到监督学习的作用，一直到满足终止条件为止
- BP 神经网络的核心思想是由后层误差推导前层误差，一层一层的反传，最终获得各层的误差估计，从而得到参数的权重值。由于权值参数的运算量过大，一般采用梯度下降法来实现
- 所谓梯度下降就是让参数向着梯度的反方向前进一段距离，不断重复，直到梯度接近零时停止。此时，所有的参数恰好达到使损失函数取得最低值的状态，为了避免局部最优，可以采用随机化梯度下降

径向基函数网络

- 径向基函数网络的隐含层是由径向基函数神经元组成，这一神经元的变换函数为径向基函数。典型的RBF网络由输入层、RBF隐层和由线性神经元组成的输出层
- 与传统的即神经网络相比，其主要区别是隐层节点中使用了径向基函数、对输入进行了高斯变换、将在原样本空间中的非线性问题，映射到高维空间中使其变得线性，然后在高维空间里用线性可分算法解决，RBF网络采用高斯函数作为核函数：

$$y = exp\left[-\left(b(x-w)\right)^2\right]$$

- RBF网络的隐层神经元自带激活函数，所以其层数可以只有一层隐层， 权重值数量更少，所以RBF网络较BP网络速度快很多
- 目前，RBF神经网络已经成功应用于非线性函数逼近、数据分类、模式识别、图像处理等方向

反馈神经网络

- 与前馈神经网络相比，反馈神经网络内部神经元之间有反馈，可以用一个无向完全图表示，包括了Hopfield 网络、BAM网络，Elman网络等
- Hopfield网络类似人类大脑的记忆原理，即通过关联的方式，将某一件事物与周围场最中的其他事物建立关联，当人们忘记了一部分信息后，可以通过场最信息回忆起来，将缺失的信息找回。通过在反馈神经网络中引入能量函数的概念，使其运行稳定性的判断有了可靠依据，由权重值派生出能量函数是从能量高的位置向能量低的位置转化，稳定点的势能比较低。基于动力学系统理论处理状态的变换，系统的稳定态可用于描述记忆
- Hopfield网络分为离散型和连续型两种网络
- 在Hopfield网络中，学习算法是基于Hebb学习规则，权值调整规则为若相邻两个神经元同时处于兴奋状态，那么他们之间的连接应增强，权值增大；反之，则权值减少

反馈神经网络

- 反馈神经网络的训练过程，主要用于实现记忆的功能，即使用能量的极小点 (吸引子)作为记忆值，一般可应用以下操作来实现训练
  - 存储:基本的记忆状态，通过权值矩阵存储
  - 验证:选代验证，直到达到稳定状态
  - 回忆:没有(失去)记忆的点，都会收敛到稳定的状态
- 以下是Hopfield网络的一个示例应用，对屏幕点阵模拟的数字进行记忆，经过克罗内克积计算之后，获得了对应数字的参数值矩阵，进行记忆效果评估时，只给出一半的点阵数字信息，通过Hopfield网络进行恢复到原始数字

```
def kroneckerSquareProduct(self, factor):
    ksProduct = np.zeros((self.N, self.N), dtype = np.float32)
    for i in xrange(0, self.N):
        ksProduct[i] = factor[i] * factor
    return ksProduct
```

反馈神经网络

```
#记忆单个数字的状态
def do_train(self, inputArray):
    # 归一化
    mean = float(inputArray.sum()) / inputArray.shape[0]
    self.W = self.W + self.kroneckerSquareProduct(inputArray - mean)/(self.N * self.N)/mean/(1-mean)
#通过记忆重构数字
def hopRun(self, inputList):
    inputArray = np.asarray(inputList, dtype = np.float32)
    matrix = np.tile(inputArray, (self.N, 1))
    matrix = self.W * matrix
    ouputArray = matrix.sum(1)
    # 归一化
    m = float(np.amin(ouputArray))
    M = float(np.amax(ouputArray))
    ouputArray = (ouputArray - m) / (M - m)
    ouputArray[ouputArray < 0.5] = 0.
    ouputArray[ouputArray > 0] = 1.
    return np.asarray(ouputArray, dtype = np.uint8)
```

反馈神经网络

- 按照之前的训练权重值恢复数字时，网络基于其他节点的值及权重的克罗内克积的和做出判定。当归一化后和小于0.5时，点阵节点设置为0，否则设置为1。运行效果如图6-4所示，图中（1）和(3)分别是缺失部分点阵信息的数字0和3，（2）和（4）分别是0和3经过Hopfield网络恢复之后的结果。可以看到，即使只提供了一半信息，Hopfield依然可以将数字识别（回忆）出来

(1)　(2)　(3)　(4)

反馈神经网络

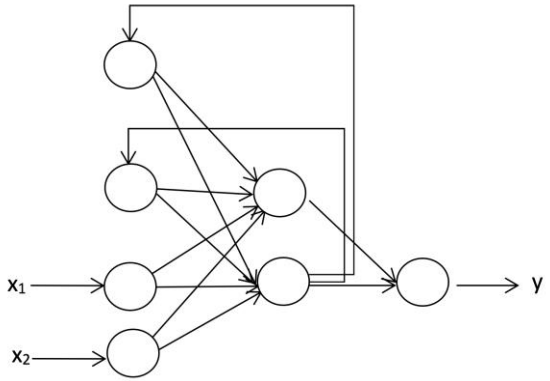
- 虽然Hopfield网络具有强大的记忆能力，但是它的缺点也比较明显：
  - 假记忆问题：只能记住有限个状态，并且当状态之间相似性较大时，或者状态的特征较少或不明显时，容易收敛到别的记忆上。
  - 存储容量限制：主要依赖极小点的数量。当两个样本距离较近时，就容易产生混淆。假设有n个神经元，那么最多可存0.15n个极小点；能够完美回忆到大部分结果只有$\frac{n}{2ln^n}$；完美回忆所有结果只有$\frac{n}{4ln^n}$。在上例中，神经元数量是25,则最多只能存储25*0.15=3.75个数字，大约可回忆的数字为3.88个，能完美回忆的数字为1.9个
  - 存在局部最优问题

反馈神经网络

- 双向联想记忆神经网络（BAM）具有非监督学习能力，网络的设计比较简单，可大规模并行处理大量数据，具有较好的实时性和容错性。此外，这种联想记忆法无需对输入向量进行预处理，省去了编码与解码的工作
- BAM是一种无条件稳定的网络，与Hopfield相比是一种特别的网络，具有输入输出节点，但是Hopfield的不足也一样存在， 即存在假记忆、存储容量限制、局部最优等问题

反馈神经网络

- Elman神经网络是一种循环神经网络，网络中存在环形结构，部分神经元的输出反馈作为输入，而这样的反馈将会出现在该网络的下一个时刻，也即这些神经元在这一时刻的输出结果，反馈回来在下一时刻重新作为输入作用于这些神经元，因此循环神经网络可以有效地应对涉及时序性的问题



反馈神经网络

- 由图中可以清晰地看出Elman网络的特点，隐层神经元的输出被反馈回来，作为一个单独的结构被定义为承接层，而承接层的数据将与输入层的数据一起作为下一时刻的输入
- Elman网络在结构上除了承接层的设置以外，其余部分与BP神经网络的结构大体相同，其隐层神经元通常采用Sigmoid函数作为激活函数，训练过程与BP神经网络相似
- Elman网络是在时间上动态的，具有内部动态反馈的功能，承接层的设置使得Elman网络能够有效应对具有时变特征的数据，在带有时序性的样本数据上有着比静态神经网络更好的预测性能

自组织神经网络

- 自组织神经网络又称Kohonen网，这一神经网络的特点是当接收到外界信号刺激时，不同区域对信号自动产生不同的响应。这种神经网络是在生物神经元上首先发现的，如果神经元是同步活跃的则信号加强，如果异步活跃则信号减弱
- MiniSom库是一种基于Python语言的Numpy实现的自组织映射(SOM)网络。下面是用SOM来实现图片量化， 合并不太重要的颜色，减少颜色数量

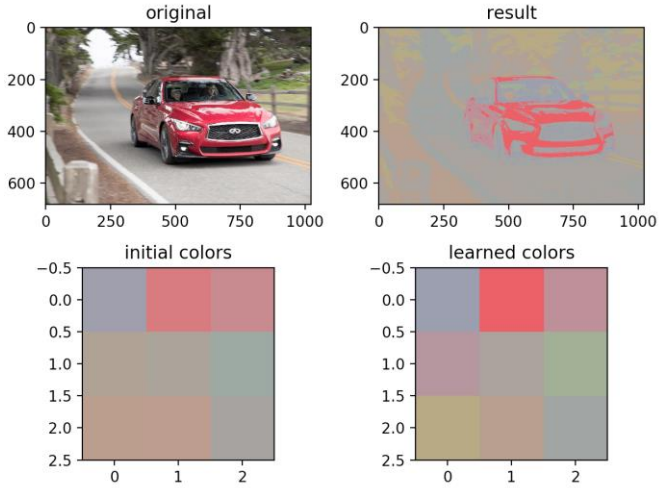
```
from minisom import MiniSom
import numpy as np
import matplotlib.pyplot as plt
img = plt.imread('car.jp2')
# 像素转换成矩阵
pixels = np.reshape(img, (img.shape[0]*img.shape[1], 3))
# 像素转换成矩阵
pixels = np.reshape(img, (img.shape[0]*img.shape[1], 3))
# SOM初始化并训练
som = MiniSom(3, 3, 3, sigma=0.1, learning_rate=0.2)  # 3x3 = 9 final colors
som.random_weights_init(pixels)
starting_weights = som.get_weights().copy()
```

自组织神经网络

```
som.train_random(pixels, 100)
qnt = som.quantization(pixels)
clustered = np.zeros(img.shape)
for i, q in enumerate(qnt):
    clustered[np.unravel_index(i, dims=(img.shape[0], img.shape[1]))] = q
#结果显示
plt.figure(1)
plt.subplot(221)
plt.title('original')
plt.imshow(img)
plt.subplot(222)
plt.title('result')
plt.imshow(clustered)
plt.subplot(223)
plt.title('initial colors')
plt.imshow(starting_weights, interpolation='none')
plt.subplot(224)
plt.title('learned colors')
plt.imshow(som.get_weights(), interpolation='none')
plt.tight_layout()
plt.show()
```

自组织神经网络

- 调用metplotlib. pmplot将结果进行可视化展示



神经网络相关概念

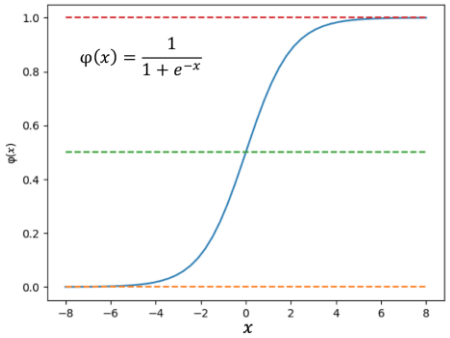
- 学习神经网络相关概念有助于理解深度学习中网络设计原理，可在模型训练过程中有的放矢地调整参数，并且这些神经网络相关概念是深度学习的基础，随着深度学习的不断演化，深入理解这些常识性理论有助于快速理解层出不穷的深度学习网络模型

激活函数

- 激活函数经常使用Sigmoid函数、tanh函数、ReLu 函数
- 激活函数通常有以下性质
  - 非线性
  - 可微性
  - 单调性
  - $f(x) \approx x$
  - 输出值范围
  - 计算简单
  - 归一化

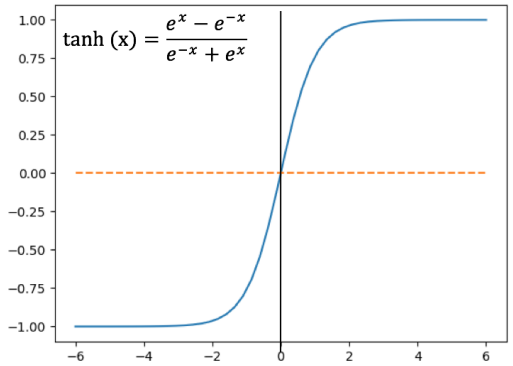
Sigmoid函数

- Sigmoid函数的优点在于输出范围有限，数据在传递的过程中不容易发散，并且其输出范围为(0,1)，可以在输出层表示概率值，如图6-8所示。Sigmoid函数的导数是非零的，很容易计算
- Sigmoid函数的主要缺点是梯度下降非常明显，且两头过于平坦，容易出现梯度消失的情况，输出的值域不对称，并非像tanh函数那样值域是-1到1



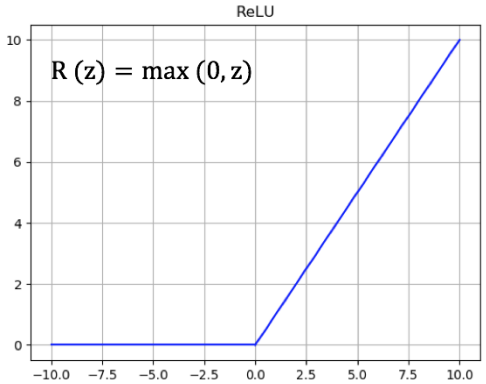
双曲正切函数(tanh)

- 双曲正切函数将数据映射到[-1,1]，解决了Sigmoid函数输出值域不对称问题。另外，它是完全可微分和反对称的，对称中心在原点。然而它的输出值域两头依旧过于平坦，梯度消失问题仍然存在。为了解决学习缓慢和梯度消失问题，可使用其更加平缓的变体，如log-log、Softsign、Symmetrical Sigmoid等



ReLU函数

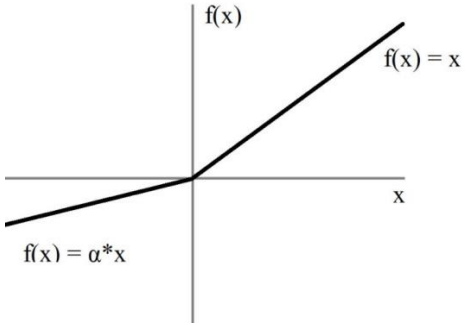
- ReLU函数是目前神经网络里常用的激活函数，由于ReLU函数是线性特点使其收敛速度比Sigmoid、Tanh更快，而且没有梯度饱和的情况出现。计算更加高效，相比于Sigmoid、Tanh函数，只需要一个阈值就可以得到激活值，不需要对输入归一化来防止达到饱和



Leaky RuLU函数

- 带泄漏修正线性神经元（ Leaky RuLU）的出现是解决“死亡神经元”的问题

$$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$$



Maxout函数

- Maxout是一种分段线性函数，理论上可以拟合任意凸函数，与其它激活函数相比，它计算k次权值，从中选择最大值作权值，所以其计算量成k倍增加。当k为2时，可看成是分成两段的线性函数，它的函数公式如下

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

激活函数

- 如何选择激活函数？
- 通常使用ReLU函数，并注意设置好学习率
- 如果存在死亡神经元的问题，就尝试Leaky ReLU或Maxout函数
- 尽量避免使用Sigmoid函数
- tahn函数大部分情况下效果不如ReLU和Maxout函数

损失函数

- 损失函数评价的是模型对样本拟合度，预测结果与实际值越接近，说明模型的拟合能力越强，对应损失函数的结果就越小；反之，损失函数的结果越大。损失函数比较大时，对应的梯度下降比较快。为了计算方便，可以采用欧式距离作损失度量标准，通过最小化实际值与估计值之间的均方误差作为损失函数，即最小平方误差准则(MSE)：

$$\min \mathrm{C}\big(\mathrm{Y}, \mathrm{G}(\mathrm{X})\big) = ||G(X) - Y||^2 = \sum_i (G(x_i) - y^i)^2$$

- 其中$G(X)$是模型根据输入矩阵X输出一个预测向量，预测值$G(X)$和真值Y的欧式距离越大、损失就越大，反之就越小,即求$||G(X) - Y||^2$的极小值。如果是批量数据，则将所有数据对应模型结果与其真实值之间的差的平方进行求和。合适的损失函数能够确保深度学习模型更好地收敛，常见的损失函数有Softmax、欧式损失、Sigmoid交叉时损失、Triplet Loss、Moon Loss、Contrastive Loss等

Softmax

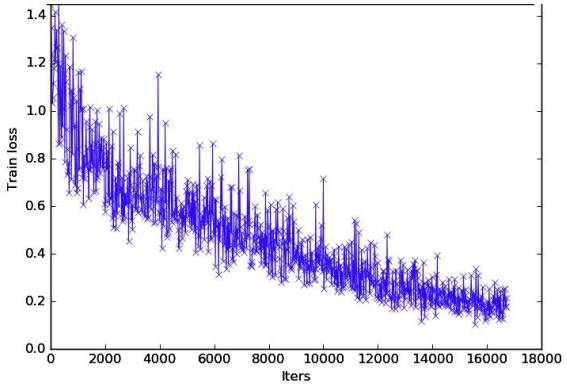
- 使用Softmax函数的好处是可以使分类问题的预测结果更加明显，不同类别之间的差距更大。在实际应用中，特别是在Tensorflow中推荐采用交叉熵与Softmax结合作为损失函数，可以避免数值不稳定的情况

交叉熵

- 目标为二分类问题，分类误差越小，则损失越小，对正负分类计算各自的损失。但是会产生梯度爆炸问题

$$\mathrm{L(w)} = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N}[y_n log\hat{y}_n + (1-y_n)\log(1-\hat{y}_n)]$$

- 交叉熵损失函数的用途主要应用在互相排斥的分类任务中
- 交叉熵也可以用于目标为[0,1]区间的回归问题



均方差损失函数

- 均方差损失函数公式如下

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2$$

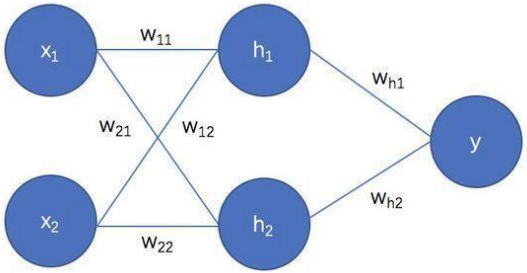
自定义函数

- 对于某些候选属性，单独将一些预测值取出或者赋予不同大小的参数。或者合并多个损失函数，实现多目标训练任务，或者在不同的情况下采用不同的损失函数

学习率

- 学习率控制每次更新参数的幅度，过高和过低的学习率都可能对模型结果带来不良影响，合适的学习率可以加快模型的训练速度
- 常见学习率调整方法
  - 基于经验的手动调整
  - 固定学习率
  - 均分分步降低策略
  - 指数级衰减
  - 多项式策略
  - AdaGrad动态调整
  - AdaDelta自动调整
  - 动量法动态调整
  - RMSProp动态调整
  - 随机梯度下降
  - Adam自动调整

神经网络算例



$$y = h_1 w_{h1} + h_2 w_{h2} = x_1 w_{11} + x_2 w_{12} + x_2 w_{21} + x_2 w_{22}$$

$$E = \frac{1}{2}(y - t)^2$$

x_1=1 , x_2=-1 , w_{11}=0.1 , w_{21}=-0.1 , w_{12}=-0.1
w_{22}=0.1 , w_{h1}=0.8 , w_{h2}=0.9 , t=0

$$h_1 = w_{11}x_1 + w_{12}x_2 = 0.2$$
$$h_2 = w_{21}x_1 + w_{22}x_2 = -0.2$$

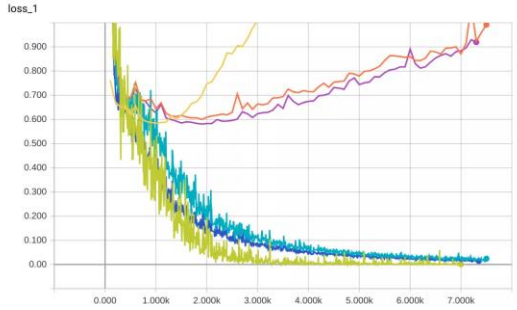
$$y = h_1 w_{h1} + h_2 w_{h2} = -0.2$$

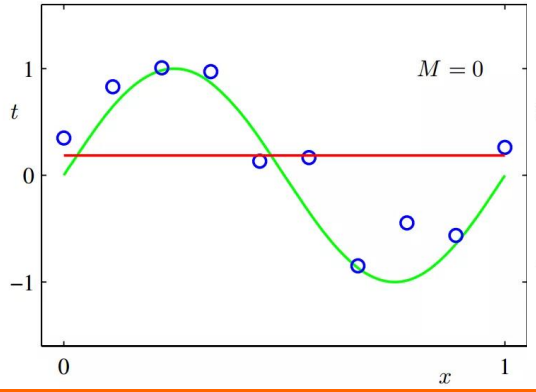
$$\frac{\partial E}{\partial w_{h1}} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_{h1}} = (y - t)h_1 = -0.04$$

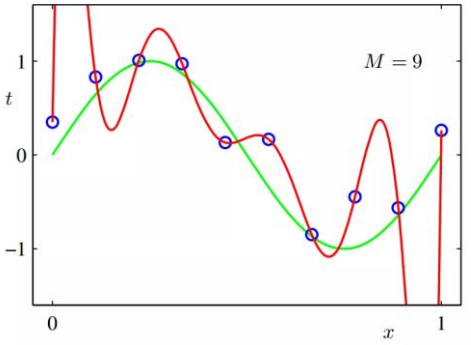
$$w_{h1} = w_{h1} - \eta \frac{\partial E}{\partial w_{h1}}$$

过拟合

- 过拟合是指模型在训练集上预测效果好，但在测试集上预测效果差
- 常用的防止过拟合的方法有
  - 参数范数惩罚
  - 数据增强
  - 提前终止
  - Bagging等集成方法
  - Dropout
  - 批正则化







模型训练中的问题

- 选择恰当的激活函数
- 权重初始化
- 学习率
- 使用Dropout正则化
- 周期 / 训练迭代次数
- 训练过程可视化

模型训练中的问题

- 在典型的Tensorflow训练过程中，可以实时将训练参数通过TensorBoard输出到文件中，并可以在浏览器中输入http://127.0.0.1:6006进行查看。其中summary_writer的作用是将参数 / 结构等写入到文件中，详细代码如下

```
with tf.Session() as session:
    init = tf.global_variables_initializer()
    session.run(init)
    summary_writer = tf.summary.FileWriter(tensorbardLogPath, session.graph)
    for epoch in range(training_epochs):
        cost_history = np.empty(shape=[1],dtype=float)
        for b in range(total_batchs):
            offset = (b * batch_size) % (train_y.shape[0] - batch_size)
            batch_x = train_x[offset:(offset + batch_size), :, :, :]
            batch_y = train_y[offset:(offset + batch_size), :]
            _, c,summary =session.run([optimizer, loss, merged],feed_dict={X: batch_x, Y : batch_y})
            cost_history = np.append(cost_history,c)
            summary_writer.add_summary(summary, epoch * training_epochs + b)
        print "Epoch: ",epoch," Training Loss: ",np.mean(cost_history)," Training Accuracy: ",session.run(accuracy,
feed_dict={X: train_x, Y: train_y})
    summary_writer.close()
    print "Testing Accuracy:", session.run(accuracy, feed_dict={X: test_x, Y: test_y})
```

模型训练中的问题

- 每一批次数据训练后会得到损失函数结果，对其进行统计，并计算出当前阶段的准确率，输出到屏幕上，在所有训练结束后再输出整体的准确率结果。如果在训练过程中发现存在异常可以直接中止训练过程，避免浪费时间

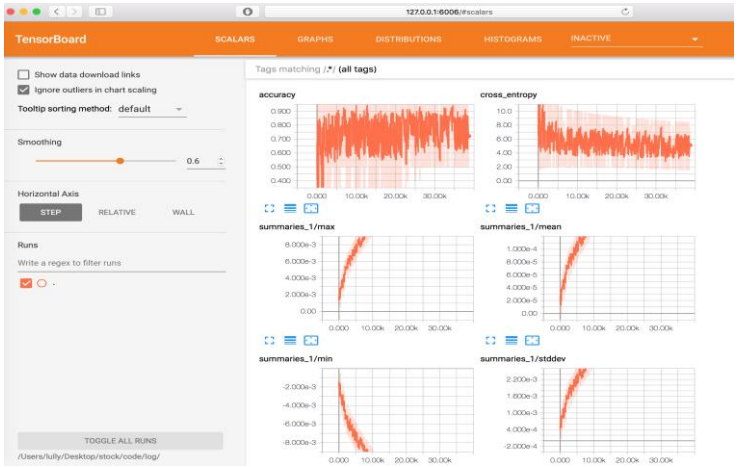
```
Epoch:  0  Training Loss:  6.47464623451  Training Accuracy:  0.616162
Epoch:  1  Training Loss:  12.3636203289  Training Accuracy:  0.676768
Epoch:  2  Training Loss:  18.0308646679  Training Accuracy:  0.666667
Epoch:  3  Training Loss:  23.6090565205  Training Accuracy:  0.666667
Epoch:  4  Training Loss:  29.1465301514  Training Accuracy:  0.666667
Epoch:  5  Training Loss:  34.6625954151  Training Accuracy:  0.666667
Epoch:  6  Training Loss:  40.1655124187  Training Accuracy:  0.666667
Epoch:  7  Training Loss:  45.6591722012  Training Accuracy:  0.666667
Epoch:  8  Training Loss:  51.1453187466  Training Accuracy:  0.666667
Epoch:  9  Training Loss:  56.6249495506  Training Accuracy:  0.666667
Epoch:  10  Training Loss:  62.0986032724  Training Accuracy:  0.666667
Epoch:  11  Training Loss:  67.5666336298  Training Accuracy:  0.666667
Epoch:  12  Training Loss:  73.0293815136  Training Accuracy:  0.666667
Epoch:  13  Training Loss:  78.4867264748  Training Accuracy:  0.666667
Epoch:  14  Training Loss:  83.938809371  Training Accuracy:  0.666667
Epoch:  15  Training Loss:  89.385931778  Training Accuracy:  0.666667
Epoch:  16  Training Loss:  94.8280973911  Training Accuracy:  0.666667
Epoch:  17  Training Loss:  100.265280128  Training Accuracy:  0.666667
Epoch:  18  Training Loss:  105.697688699  Training Accuracy:  0.666667
Epoch:  19  Training Loss:  111.12521441  Training Accuracy:  0.666667
Testing Accuracy: 0.688889
```

模型训练中的问题

- 在命令行中输入以下命令启动TensorBoard,可以查看之前通过summary_writer写入的参数值

```
$ tensorboard --logdir=/Users/lully/Desktop/stock/de/log/
```

- 在浏览器中输入本机和端口（默认为6006），可以实时查看之前定义并写入的参数和指标



模型训练中的问题

- 其中的变量如何记录呢？以accuracy对应的变化曲线图为例，使用如下代码

```
with tf.name_scope('train'):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(loss)
    correct_prediction = tf.equal(tf.argmax(y_,1), tf.argmax(Y,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy', accuracy)
```

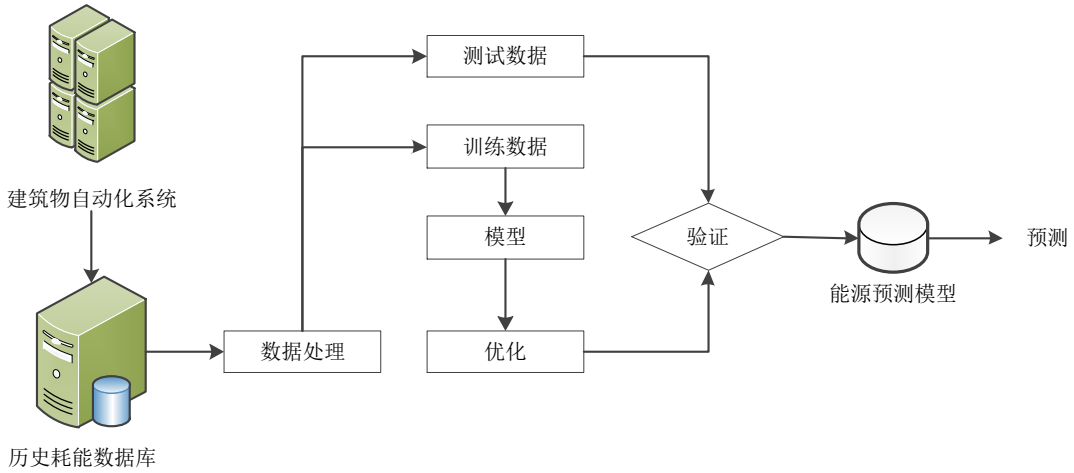
- 其它变量如cross_entropy等的记录方法与此类似，SCALARS面板中主要用于记录诸如准确率、损失和学习率等单个值的变化趋势。此外，还支持对TensorFlow中计算图的结构进行可视化，显示每个节点的计算时间和内存使用情况等。如果要显示内存等信息，可以使用如下代码，主要是在sess.run()中加入options和run_metadata参数，然后就可以在GRAPHS页面中查看对应节点的计算时间或内存信息，使用颜色深浅来表示

神经网络效果评价

- 用于分类的模型评价以准确率(Accuracy)、精确率(Precision)、召回率(Recall)、F1分值(F1 Score)为主，辅以ROC、AUC并结合实际应用进行结果评价
- 如果神经网络用于聚类，数据源并没有进行标记，那么其模型结果的评价按照聚类算法的标准来操作，如RMSSTD、R Square、SRP等
- 随着机器学习在不同领域中的应用，其评价方式需要与实际业务相结合，通过确定目标要求来定量设计评价标准，例如在目标检测等方面使用平均曲线下面积(mean Average Precision, mAP)指标进行衡量识别准确性

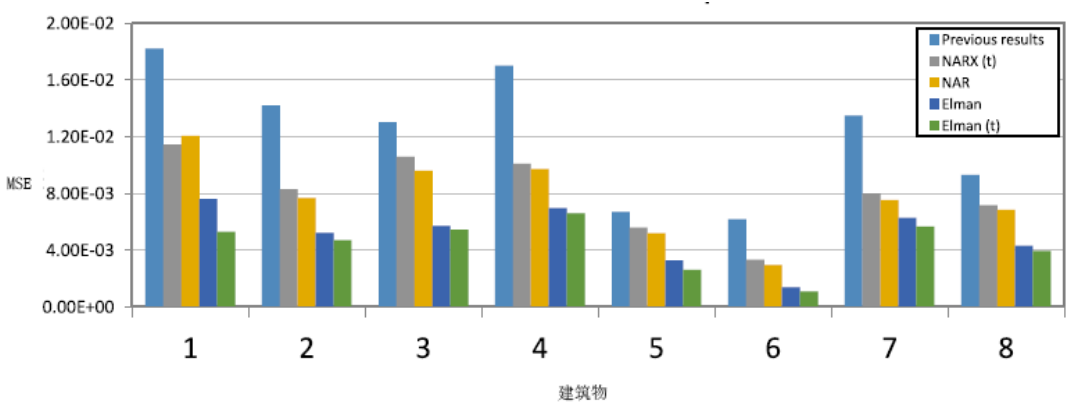
神经网络的应用

- 基于Elman神经网络的能源消耗预测



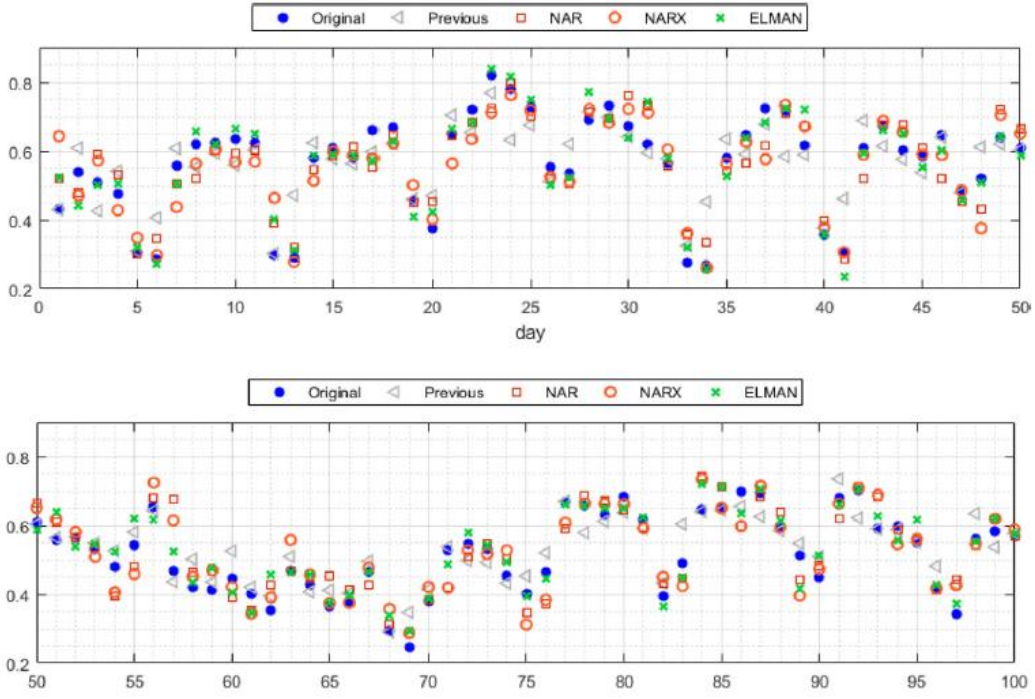
基于Elman神经网络的能源消耗预测

- 预测实验结果图



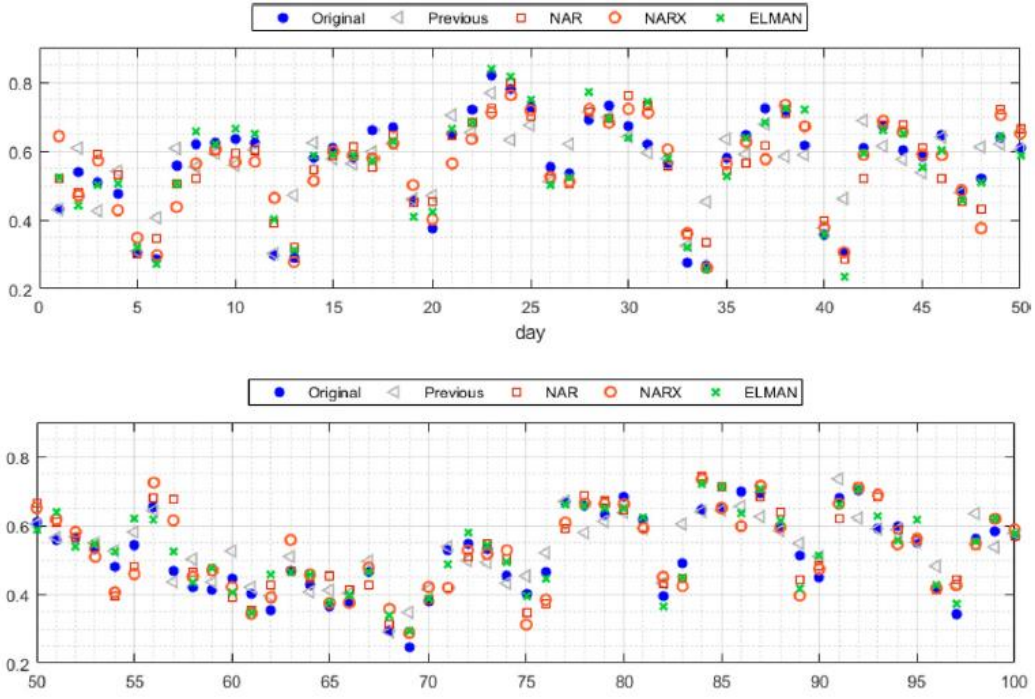
基于Elman神经网络的能源消耗预测

- 100天预测结果走势比对图



基于Elman神经网络的能源消耗预测

- 100天预测结果走势比对图



基于神经网络(多层感知器)识别手写数字

- 数据集为经典的MNIST
- 加载数据

```
import tensorflow as tf
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./mnist_data/", one_hot=True)
```

基于神经网络(多层感知器)识别手写数字

- 定义学习率、迭代次数、批大小、批数量（总样本数除以批大小）等参数，设置输入层大小为784,即将28x28的像素展开为一维行向量（一个输入图片784个值）。第一层和第二层神经元数量均为256，输出层的分类类别为0～9的数字，即10个类别

```
learning_rate = 0.005
training_epochs = 20
batch_size = 100
batch_count = int(mnist.train.num_examples/batch_size)
n_hidden_1 = 256
n_hidden_2 = 256
n_input = 784
n_classes = 10 # (0-9 数字)
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])
```

基于神经网络(多层感知器)识别手写数字

- 使用tf.random_normal()生成模型权重值参数矩阵和偏置值参数，并将其分别存储于weights和biases变量中，并定义多层感知机的神经网络模型。代码如下

```
weights = {
    'weight1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'weight2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'bias1': tf.Variable(tf.random_normal([n_hidden_1])),
    'bias2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
def multilayer_perceptron_model(x):
    layer_1 = tf.add(tf.matmul(x, weights['weight1']), biases['bias1'])
    layer_2 = tf.add(tf.matmul(layer_1, weights['weight2']), biases['bias2'])
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

基于神经网络(多层感知器)识别手写数字

- 使用输入变量X初始化模型，定义损失函数为交叉熵，采用梯度下降法作为优化器（除此之外还可选MomentumOptimizer、AdagradOptimizer、AdamOptimizer等，见注释部分），并对模型中tf.placeholder定义的各参数初始化，代码如下

```
logits = multilayer_perceptron_model(X)
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
#optimizer = tf.train.MomentumOptimizer(learning_rate,0.2)
#optimizer = tf.train.AdagradOptimizer(learning_rate)
#optimizer = tf.train.AdamOptimizer(learning_rate)
train_op = optimizer.minimize(loss_op)
init = tf.global_variables_initializer()#参数初始化
```

基于神经网络(多层感知器)识别手写数字

- 将训练集样本输入模型进行训练，并计算每个批次的平均损失，在每次迭代时输出模型的平均损失，代码如下

```
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(training_epochs):
        avg_cost = 0.
        for i in range(batch_count):
            train_x, train_y =  mnist.train.next_batch(batch_size)
            _, c = sess.run([train_op, loss_op], feed_dict={X: train_x, Y: train_y})
            avg_cost += c / batch_count
        print("Epoch:", '%02d' % (epoch+1), "avg cost={:.6f}".format(avg_cost))
```

基于神经网络(多层感知器)识别手写数字

- 模型训练完成，使用测试集样本对其评估，并计算其精确率，代码如下

```
pred = tf.nn.softmax(logits)  # Apply softmax to logits
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print("Accuracy:", accuracy.eval({X: mnist.test.images, Y: mnist.test.labels}))
```

- 模型的Accuracy结果为87.8%


THANK
YOU