人工智能实践：Tensorflow笔记

曹 健

北京大学

软件与微电子学院

本讲目标：学会神经网络优化过程，使用正则化减少过拟合，使用优化器更新网络参数

- 预备知识
- 神经网络复杂度
- 指数衰减学习率
- 激活函数
- 损失函数
- 欠拟合与过拟合
- 正则化减少过拟合
- 优化器更新网络参数 用python编写反向传播优化器，实现参数更新

- 预备知识

tf.where()

✓ 条件语句真返回A，条件语句假返回B

tf.where(条件语句，真返回A，假返回B)

```
a=tf.constant([1,2,3,1,1])

b=tf.constant([0,1,3,4,5])

c=tf.where(tf.greater(a,b), a, b) # 若a>b，返回a对应位置的元素，否则返回b对应位置的元素

print("c:",c)
```

运行结果：
c： tf.Tensor([1 2 3 4 5], shape=(5,), dtype=int32)

源码:p4_where.py

np.random.RandomState.rand()

✓ 返回一个[0,1)之间的随机数

np.random.RandomState.rand(维度)　　　　#维度为空，返回标量

```
import numpy as np

rdm=np.random.RandomState(seed=1) #seed=常数每次生成随机数相同

a=rdm.rand()     # 返回一个随机标量

b=rdm.rand(2,3)  # 返回维度为2行3列随机数矩阵

print("a:",a)

print("b:",b)
```

运行结果：

```
a: 0.417022004702574
b: [[7.20324493e-01 1.14374817e-04 3.02332573e-01]
    [1.46755891e-01 9.23385948e-02 1.86260211e-01]]
```

源码：p5_RandomState.py

np.vstack()

✓ 将两个数组按垂直方向叠加

np.vstack(数组1，数组2)

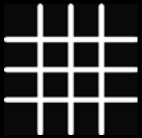
```
import numpy as np
a = np.array([1,2,3])
b = np.array([4,5,6])
c = np.vstack((a,b))
print("c:\n",c)
```

运行结果：

```
c:
 [[1 2 3]
 [4 5 6]]
```

源码：p6_vstack.py

np.mgrid[ ]   .ravel( )   np.c_[ ]



这三个函数经常一起使用，生成网格坐标点

- ✓ **np.mgrid[ ]** 返回若干组维度相同的等差数组 　　　　**[起始值 结束值)**

  np.mgrid[起始值 : 结束值 : 步长， 起始值 : 结束值 : 步长 ， …]

行数由mgrid的第一个参数决定，
列数由mgrid的第二个参数决定

- ✓ **x.ravel( ) 将x变为一维数组， “把▪前变量拉直”**

- ✓ **np.c_[ ] 使返回的间隔数值点配对**

  np.c_[数组1，数组2， …]

```
import numpy as np
x, y = np.mgrid [1:3:1, 2:4:0.5]
grid = np.c_[x.ravel(), y.ravel()]
print("x:",x)
print("y:",y)
print('grid:\n', grid)
```

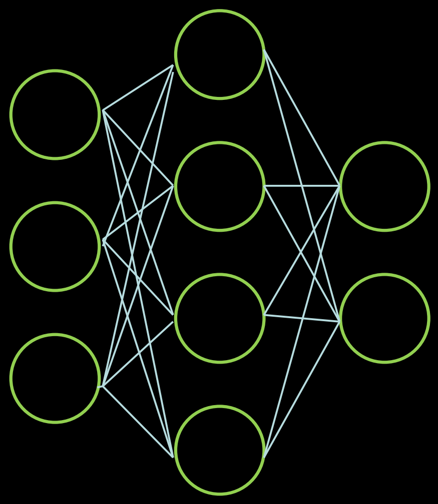
运行结果:

```
x = [[1. 1. 1. 1.]
     [2. 2. 2.  2.]]
y = [[2. 2.5 3. 3.5]
     [2. 2.5 3. 3.5]]
grid:
 [[1.  2. ]
 [1.  2.5]
 [1.  3. ]
 [1.  3.5]
 [2.  2. ]
 [2.  2.5]
 [2.  3. ]
 [2.  3.5]]
```

源码：p7_mgrid.py

神经网络（NN）复杂度

✓ **NN复杂度：多用NN层数和NN参数的个数表示**



输入层　隐藏层　输出层

空间复杂度：

✓ **层数 = 隐藏层的层数 + 1个输出层**

左图为**2**层**NN**

每个神经元有一个偏置项b

✓ **总参数 = 总w + 总b**

左图**3x4+4 + 4x2+2 = 26**

第**1**层　第**2**层

时间复杂度：

✓ **乘加运算次数**

左图 **3x4 + 4x2 = 20**

第**1**层　第**2**层

学习率

表征了参数每次更新的幅度

$$w_{t+1} = w_t - lr * \frac{\partial loss}{\partial w_t}$$

更新后的参数　　当前参数　　学习率　　损失函数的梯度（偏导数）

eg：损失函数 loss = (w + 1)^2

$$\frac{\partial loss}{\partial w} = 2w + 2$$

参数w初始化为5，学习率为0.2 则

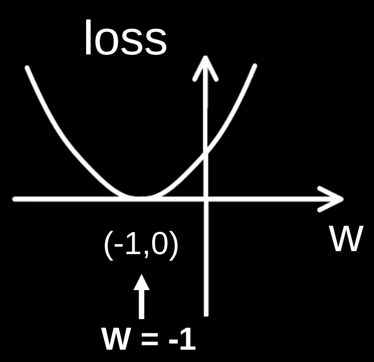
1次　　参数w：5　　　5 - 0.2 * (2 * 5 + 2) = 2.6

2次　　参数w：2.6　　2.6 - 0.2 * (2 * 2.6 + 2) = 1.16

3次　　参数w：1.16　　1.16 – 0.2 * (2 * 1.16 + 2) = 0.296

4次　　参数w：0.296

……



源码：class1\p13_backpropagation.py　　lr=0.001过慢，lr=0.999不收敛

指数衰减学习率

根据当前迭代次数，动态改变学习率的值
一般写在for循环中

可以先用较大的学习率，快速得到较优解，然后逐步减小学习率，使模型在训练后期稳定。

指数衰减学习率 = 初始学习率 * 学习率衰减率（当前轮数 / 多少轮衰减一次）

```
epoch = 40
LR_BASE = 0.2
LR_DECAY = 0.99
LR_STEP = 1

for epoch in range(epoch):
    lr = LR_BASE * LR_DECAY ** (epoch / LR_STEP)
    with tf.GradientTape() as tape:
        loss = tf.square(w + 1)
    grads = tape.gradient(loss, w)

    w.assign_sub(lr * grads)
    print("After %s epoch,w is %f,loss is %f,lr is %f"
        % (epoch, w.numpy(), loss,lr))
```

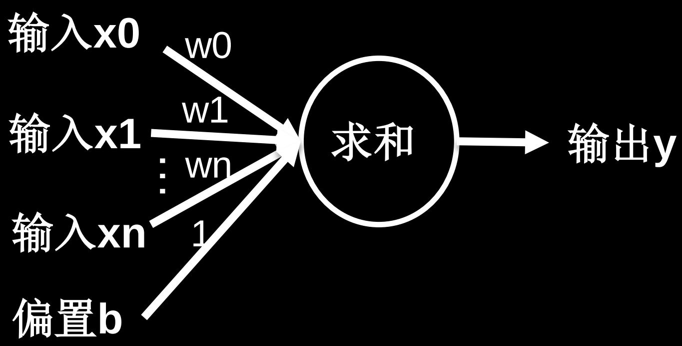
源码：class1\p13_backpropagation.py → class2\p10_backpropagation_decaylr.py

```
After 0 epoch,w is 2.600000,loss is 36.000000,lr is 0.200000
After 1 epoch,w is 1.174400,loss is 12.959999,lr is 0.198000
After 2 epoch,w is 0.321948,loss is 4.728015,lr is 0.196020
After 3 epoch,w is -0.191126,loss is 1.747547,lr is 0.194060
After 4 epoch,w is -0.501926,loss is 0.654277,lr is 0.192119
After 5 epoch,w is -0.691392,loss is 0.248077,lr is 0.190198
After 6 epoch,w is -0.807611,loss is 0.095239,lr is 0.188296
After 7 epoch,w is -0.879339,loss is 0.037014,lr is 0.186413
After 8 epoch,w is -0.923874,loss is 0.014559,lr is 0.184549
After 9 epoch,w is -0.951691,loss is 0.005795,lr is 0.182703
After 10 epoch,w is -0.969167,loss is 0.002334,lr is 0.180876
```

源码：p10_backpropagation_decaylr.py

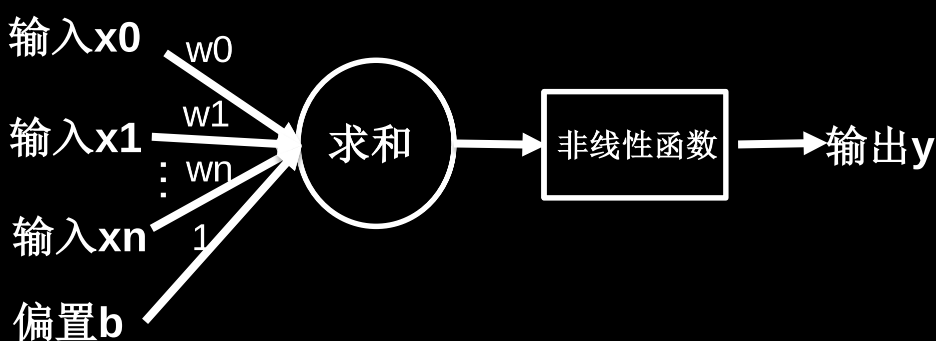
激活函数

对于线性函数，即使有多层神经元首尾相接，构成深层神经网络，依旧是线性组合，
模型的表达力不够



$$y = x * w + b$$

简化模型



$$y = \boxed{f}(x * w + b)$$

激活函数

大大提升了模型的表达力
使得多层神经网络不再是输入x的线性组合，
神经网络可以随层数的增加提升表达能力了

MP模型

激活函数

✓优秀的激活函数：

- **非线性：** 激活函数非线性时，多层神经网络可逼近所有函数
- **可微性：** 优化器大多用梯度下降更新参数
- **单调性：** 当激活函数是单调的，能保证单层网络的损失函数是凸函数
- **近似恒等性：** **f(x)≈x**当参数初始化为随机小值时，神经网络更稳定

✓激活函数输出值的范围：

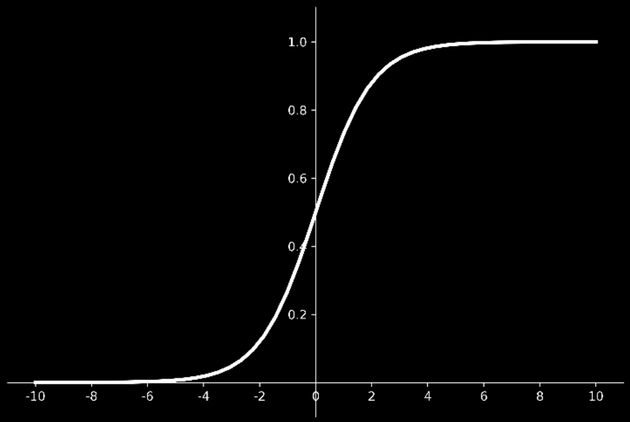
- 激活函数输出为<u>有限值</u>时，基于梯度的优化方法更<u>稳定</u>
- 激活函数输出为<u>无限值</u>时，建议<u>调小学习率</u>

激活函数

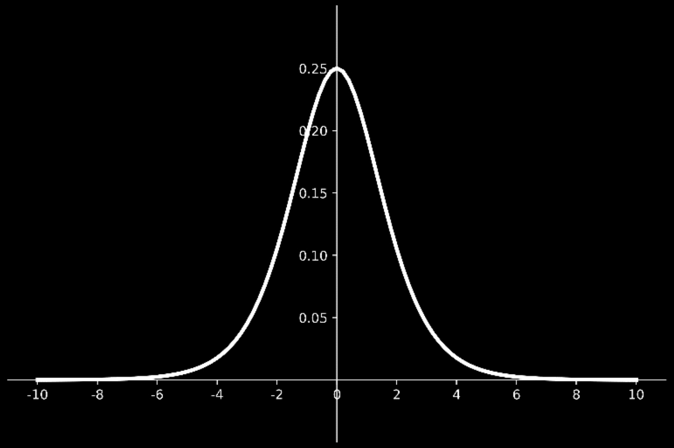
✓ **Sigmoid函数**

tf.nn. sigmoid(x)

$$f(x) = \frac{1}{1+e^{-x}}$$



函数图像



导数图像

特点

（1）易造成梯度消失

（2）输出非0均值，收敛慢

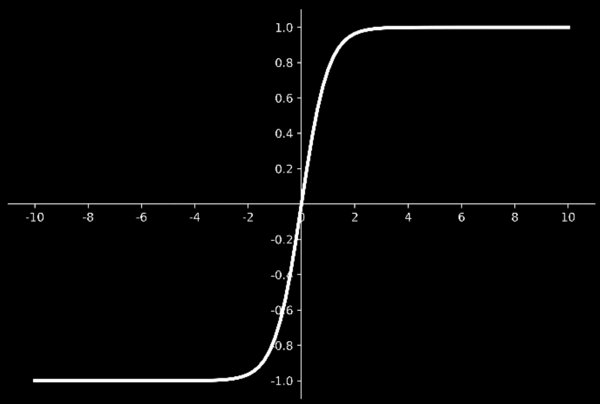
（3）幂运算复杂，训练时间长

激活函数

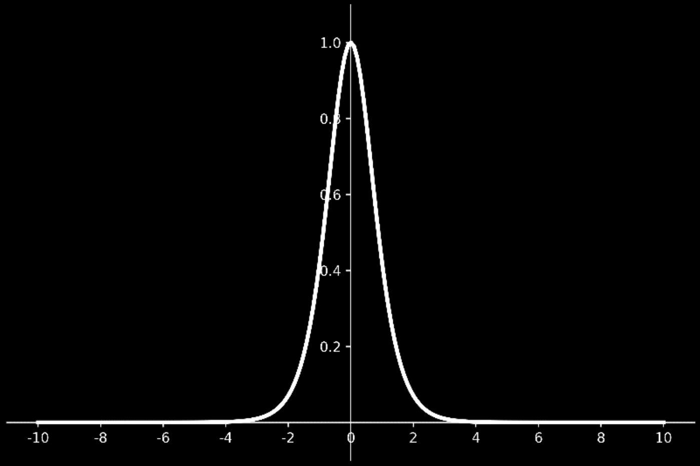
✓ **Tanh函数**

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

tf.math. tanh(x)



函数图像



导数图像

特点

（1）输出是0均值

（2）易造成梯度消失

（3）幂运算复杂，训练时间长

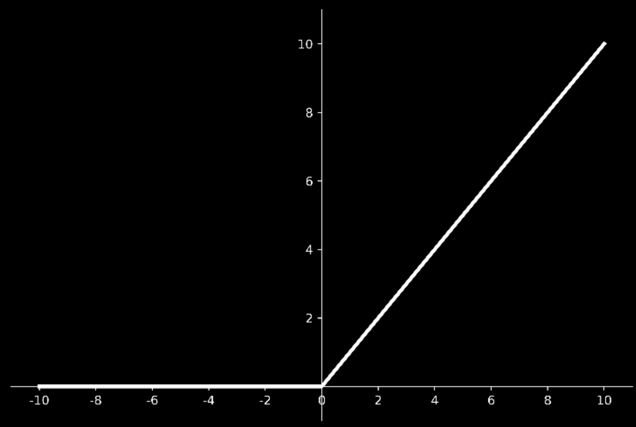
激活函数

✓ Relu函数

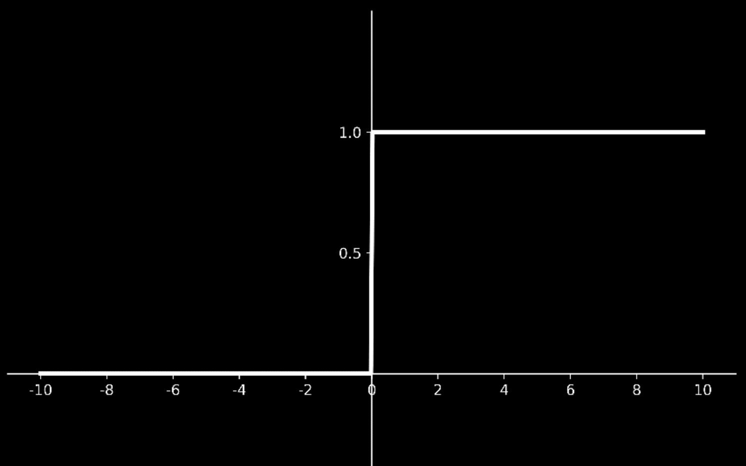
$$f(x) = \max(x, 0)$$

$$= \begin{cases} 0 & x < 0 \\ x & x >= 0 \end{cases}$$

tf.nn.relu(x)



函数图像



导数图像

优点：

（1） 解决了梯度消失问题 (在正区间)

（2） 只需判断输入是否大于0，计算速度快

（3） 收敛速度远快于sigmoid和tanh

缺点：

（1） 输出非0均值，收敛慢

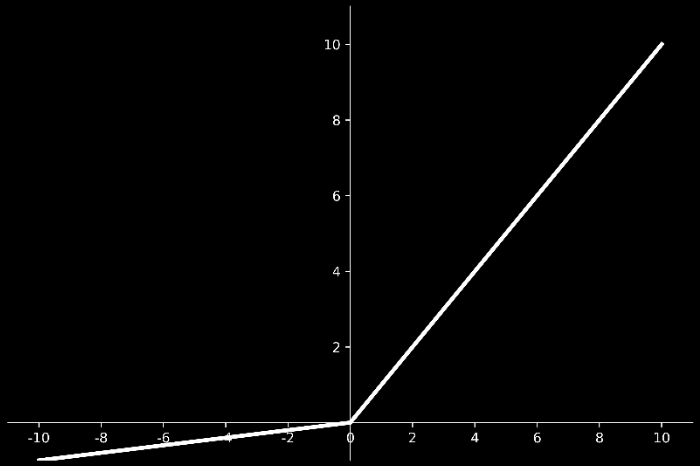
（2） Dead RelU问题：某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。

激活函数

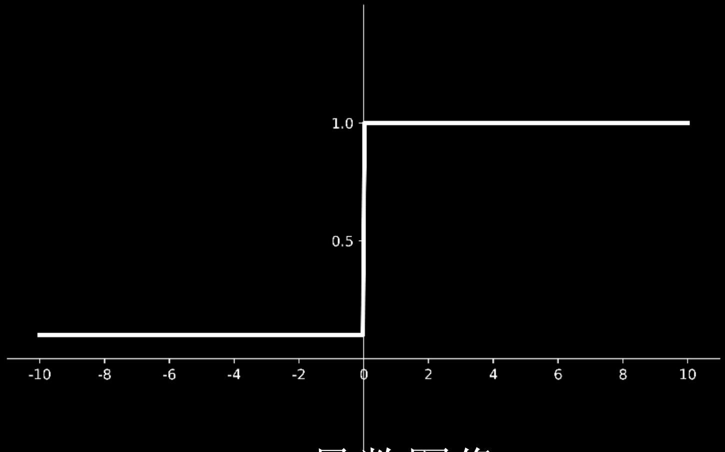
✓ Leaky Relu函数

$$f(x) = \max(\alpha x, x)$$

tf.nn.leaky_relu(x)



函数图像



导数图像

理论上来讲，Leaky Relu有Relu的所有优点，外加不会有Dead Relu问题，但是在实际操作中，并没有完全证明Leaky Relu总是好于Relu。

激活函数

对于初学者的建议：

✓ 首选relu激活函数；

✓ 学习率设置较小值；

✓ 输入特征标准化，即让输入特征满足以0为均值，1为标准差的正态分布；

✓ 初始参数中心化，即让随机生成的参数满足以0为均值，$\sqrt{\frac{2}{\text{当前层输入特征个数}}}$ 为标准差的正态分布。

√ 损失函数（loss）：预测值（y）与已知答案（y_）的差距

NN优化目标： loss最小 ➡ mse (Mean Squared Error) 均方误差 / 自定义 / ce (Cross Entropy) 交叉熵

√ 均方误差 mse： MSE(y_, y)= $\frac{\sum_{i=1}^{n}(y-y\_)^2}{n}$

loss_mse = tf.reduce_mean(tf.square(y_ - y))

预测酸奶日销量y，x1、x2是影响日销量的因素。

建模前，应预先采集的数据有：每日x1、x2和销量y_（即已知答案，最佳情况：产量=销量）

拟造数据集X,Y_： y_ = x1 + x2 噪声：-0.05 ~ +0.05 拟合可以预测销量的函数

源码：p19_mse.py

✓ 自定义损失函数：

使用均方误差作为损失函数，默认认为销量预测的多了或者少了，损失是一样的，真实情况并非如此

如预测商品销量，预测多了，损失成本；预测少了，损失利润。
若利润 ≠ 成本，则mse产生的loss无法利益最大化。

自定义损失函数 $loss(y\_, y) = \sum_n f(y\_, y)$

$y\_$：标准答案 数据集的

y：预测答案 计算出的

$$f(y\_, y) = \begin{cases} \text{PROFIT} * (y\_ - y) & y < y\_ \\ \text{COST} * (y - y\_) & y >= y\_ \end{cases}$$

预测的 y 少了，损失利润(PROFIT)
预测的 y 多了，损失成本(COST)

loss_zdy= $tf.reduce\_sum(\boxed{tf.\text{where}(tf.greater(y, y\_)}, \text{COST}(y - y\_), \text{PROFIT}(y\_ - y)))$

$\boxed{y > y\_ \ ?}$ 真 → COST，假 → PROFIT

如：预测酸奶销量，酸奶成本（COST）1元，酸奶利润（PROFIT）99元。

预测少了损失利润99元，大于预测多了损失成本1元。

预测少了损失大，希望生成的预测函数往多了预测。

源码：p20_custom.py

损失函数

✓ 交叉熵损失函数**CE (Cross Entropy)**：表征两个概率分布之间的距离

$$H(y\_,\ y) = -\sum y\_ * \ln y$$

eg. 二分类 已知答案**$y\_=(1, 0)$**　预测**$y_1=(0.6, 0.4)$**　**$y_2=(0.8, 0.2)$**

哪个更接近标准答案?

$H_1((1,0),(0.6,0.4)) = -(1*\ln 0.6 + 0*\ln 0.4) \approx -(-0.511 + 0) = 0.511$

$H_2((1,0),(0.8,0.2)) = -(1*\ln 0.8 + 0*\ln 0.2) \approx -(-0.223 + 0) = 0.223$

因为**$H_1 > H_2$**，所以**y_2**预测更准

tf.losses.categorical_crossentropy(y_，y)

损失函数

```
loss_ce1=tf.losses.categorical_crossentropy([1,0],[0.6,0.4])
loss_ce2=tf.losses.categorical_crossentropy([1,0],[0.8,0.2])
print("loss_ce1:", loss_ce1)
print("loss_ce2:", loss_ce2)
```

运行结果：

```
loss_ce1: tf.Tensor(0.5108256, shape=(), dtype=float32)
loss_ce2: tf.Tensor(0.22314353, shape=(), dtype=float32)
```

源码：p22_ce.py

损失函数

在执行分类问题时，通常先用softmax函数，让输出结果符合概率分布，再求交叉熵损失函数

softmax与交叉熵结合

✓ 输出先过softmax函数，再计算y与y_的交叉熵损失函数。

tf.nn.softmax_cross_entropy_with_logits(y_， y)

TensorFlow给出了一个可同时计算概率分布和交叉熵的函数

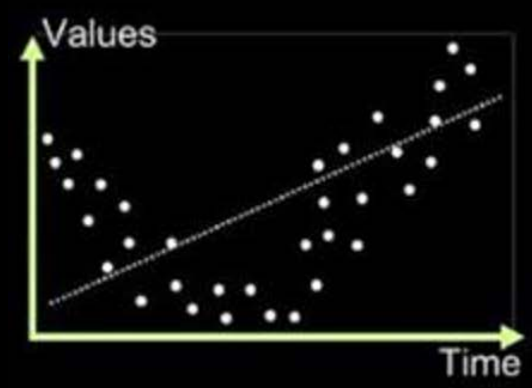
```
y_ = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 0], [0, 1, 0]])
y  = np.array([[12, 3, 2], [3, 10, 1], [1, 2, 5], [4, 6.5, 1.2], [3, 6, 1]])
y_pro = tf.nn.softmax(y)
loss_ce1 = tf.losses.categorical_crossentropy(y_, y_pro)
loss_ce2 = tf.nn.softmax_cross_entropy_with_logits(y_, y)

print('分步计算的结果:\n', loss_ce1)
print('结合计算的结果:\n', loss_ce2)
```

```
运行结果：
分步计算的结果:
 tf.Tensor(
[1.68795487e-04 1.03475622e-03 6.58839038e-02 2.58349207e+00
 5.49852354e-02], shape=(5,), dtype=float64)
结合计算的结果:
 tf.Tensor(
[1.68795487e-04 1.03475622e-03 6.58839038e-02 2.58349207e+00
 5.49852354e-02], shape=(5,), dtype=float64)
```

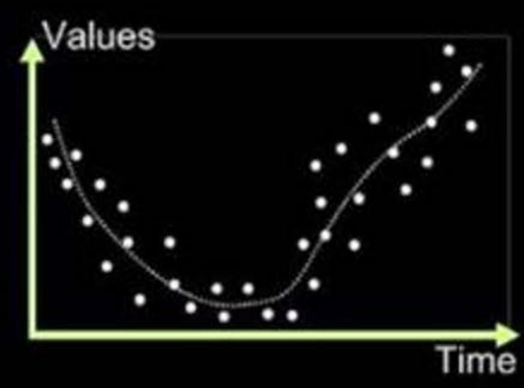
源码：p23_softmaxce.py

欠拟合与过拟合



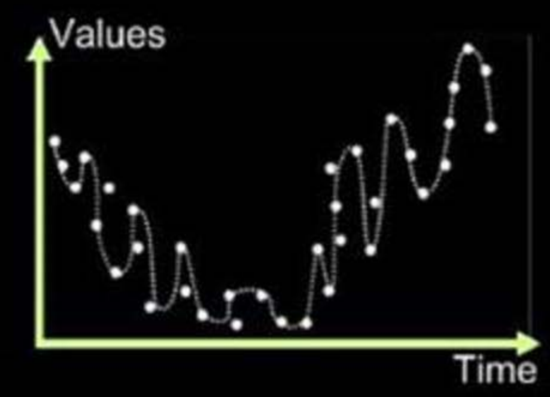
欠拟合

曲线没能有效表征数据点



正确拟合

曲线表达了点的分布



过拟合

曲线虽然表达出了每一个数据点，但模型泛化性弱

欠拟合与过拟合

✓ 欠拟合的解决方法：

增加输入特征项 给网络更多维度的输入特征

扩展网络规模 增加网络参数 增加网络深度，提升模型表达力

减少正则化参数

✓ 过拟合的解决方法：

数据清洗 减少数据集中的噪声，使数据集更纯净

增大训练集 让模型见到更多数据

采用正则化

增大正则化参数

欠拟合与过拟合

- ✓ 欠拟合的解决方法：
  - 增加输入特征项
  - 增加网络参数
  - 减少正则化参数
- ✓ 过拟合的解决方法：
  - 数据清洗
  - 增大训练集
  - 采用正则化 一种通用的、有效的方法
  - 增大正则化参数

✓ 正则化缓解过拟合

正则化在损失函数中引入模型复杂度指标，利用给W加权值，弱化了训练数据的噪声（一般不正则化b）

参数的权重

loss = loss(y与y_) + REGULARIZER * loss(w)

预测结果与正确结果之间的差距

模型中所有参数的损失函数
如：交叉熵、均方误差

用超参数REGULARIZER
给出参数w在总loss中的
比例，即正则化的权重

需要正则化的参数

$$loss_{L1}(w) = \sum_i |w_i|$$

$$loss_{L2}(w) = \sum_i |w_i^2|$$

✓ 正则化的选择

L1正则化大概率会使很多参数变为零，因此该方法可通过稀疏参数，即减少参数的数量，降低复杂度。

L2正则化会使参数很接近零但不为零，因此该方法可通过减小参数值的大小降低复杂度。

可以有效缓解数据集中因噪声引起的过拟合

✓ 正则化缓解过拟合

```
with tf.GradientTape() as tape:  # 记录梯度信息

    h1 = tf.matmul(x_train, w1) + b1  # 记录神经网络乘加运算
    h1 = tf.nn.relu(h1)
    y = tf.matmul(h1, w2) + b2

    # 采用均方误差损失函数mse = mean(sum(y-out)^2)
    loss_mse = tf.reduce_mean(tf.square(y_train - y))
    # 添加l2正则化
    loss_regularization = []
    loss_regularization.append(tf.nn.l2_loss(w1))
    loss_regularization.append(tf.nn.l2_loss(w2))
    loss_regularization = tf.reduce_sum(loss_regularization)
    loss = loss_mse + 0.03 * loss_regularization #REGULARIZER = 0.03

# 计算loss对各个参数的梯度
variables = [w1, b1, w2, b2]
grads = tape.gradient(loss, variables)
```

✓ 正则化缓解过拟合

class2\dot.csv

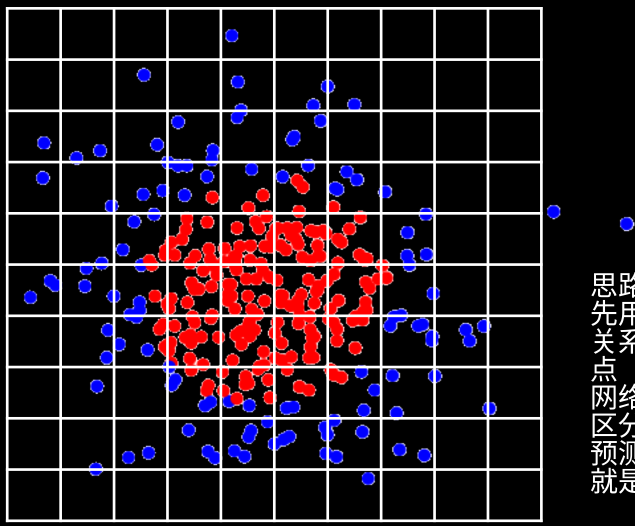
x1	x2	y_c
-0.41676	-0.05627	1
-2.1362	1.640271	0
-1.79344	-0.84175	0
0.502881	-1.24529	1
-1.05795	-0.90901	1
0.551454	2.292208	0
0.041539	-1.11793	1
0.539058	-0.59616	1
-0.01913	1.175001	1
-0.74787	0.009025	1
-0.87811	-0.15643	1
0.25657	-0.98878	1
-0.33882	-0.23618	1
-0.63766	-1.18761	1
-1.42122	-0.1535	0
-0.26906	2.231367	0
-2.43477	0.112727	0
0.370445	1.359634	1
0.501857	-0.84421	1
……		

0.380472 -0.21714 ?

把x1和x2分别作为横纵坐标，可视化数据

标签1 - 红色
标签0 - 蓝色

让神经网络画出一条线区分红色点和蓝色点



思路：
先用神经网络拟合出输入特征x1、x2与标签的函数关系，然后生成网格覆盖这些点，把这些网格的交点（也就是横纵坐标）作为输入送入训练好的神经网络，神经网络会为每个坐标输出一个预测值，要区分输出偏向1还是偏向0，可以把神经网络输出的预测值为0.5的线标出颜色，这条线就是0和1（也就是红点和蓝点）的区分线了

源码：p29_regularizationfree.py p29_regularizationcontain.py

神经网络参数优化器

神经网络是基于连接的人工智能，当网络结构固定后，不同参数选取对模型的表达力影响很大

优化器是引导神经网络更新参数的工具

待优化参数w，损失函数$loss$，学习率lr，每次迭代一个batch，t表示当前batch迭代的总次数：

训练时为了提高效率，数据集中的数据不是一次喂入神经网络一组，而是以batch为单位批量喂入神经网络，每个batch通常包含2^n组数据

更新参数分为四步完成：

1. **计算t时刻损失函数关于当前参数的梯度**$g_t = \nabla loss = \frac{\partial loss}{\partial (w_t)}$
2. **计算t时刻一阶动量**m_t**和二阶动量**V_t
3. **计算t时刻下降梯度：**$\eta_t = lr \cdot m_t / \sqrt{V_t}$
4. **计算t+1时刻参数：**$w_{t+1} = w_t - \eta_t = w_t - lr \cdot m_t / \sqrt{V_t}$

参数更新公式

一阶动量：与梯度相关的函数

二阶动量：与梯度平方相关的函数

不同优化器实质上只是定义了不同的一阶动量和二阶动量公式

优化器

最常用的是随机梯度下降

✓**SGD（无momentum），常用的梯度下降法。**

$$m_{\mathrm{t}} = g_{\mathrm{t}} \qquad V_{\mathrm{t}} = 1$$

$$\eta_{\mathrm{t}} = lr \cdot m_{\mathrm{t}} / \sqrt{V_t} = lr \cdot g_t$$

$$\begin{aligned} w_{\mathrm{t+1}} &= w_t - \eta_t \\ &= w_t - lr \cdot m_t / \sqrt{V_t} = w_t - lr \cdot g_t \end{aligned}$$

$$\boxed{\mathrm{w}_{\mathrm{t+1}} = w_t - lr * \frac{\partial loss}{\partial w_t}}$$

优化器

✓**SGD（无momentum）**

```
# sgd
w1.assign_sub(lr * grads[0])  # 参数w1自更新
b1.assign_sub(lr * grads[1])  # 参数b自更新
```

源码：p32_sgd.py

优化器

✓**SGDM（含momentum的SGD），在SGD基础上增加一阶动量。**

表示各时刻梯度方向的指数滑动平均值

β 是个接近1的超参数，经验值是0.9

$$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot g_t \qquad V_t = 1$$

上一时刻的一阶动量

当前时刻梯度
（用with结构直接算出损失函数对各个参数的偏导数）

$$\eta_t = lr \cdot m_t / \sqrt{V_t} = lr \cdot m_t$$
$$= lr \cdot (\beta \cdot m_{t-1} + (1-\beta) \cdot g_t)$$

$$w_{t+1} = w_t - \eta_t$$
$$= w_t - lr \cdot (\beta \cdot m_{t-1} + (1-\beta) \cdot g_t)$$

源码：p33_sgdm.py

优化器

✓**SGDM（含momentum的SGD）**

$$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot g_t \qquad V_t = 1$$

```
m_w, m_b = 0, 0
beta = 0.9
# sgd-momentun
m_w = beta * m_w + (1 - beta) * grads[0]
m_b = beta * m_b + (1 - beta) * grads[1]
w1.assign_sub(lr * m_w)
b1.assign_sub(lr * m_b)
```

第0时刻的一阶动量是0

自减操作实现参数w和b的自更新

源码：p34_sgdm.py

优化器

可以对模型中的每个参数分配自适应学习率
Adagrad的一阶动量和SGD一样，是当前的梯度
二阶动量是从开始到现在梯度平方的累计和

✓**Adagrad，在SGD基础上增加二阶动量**

$$m_t = g_t \qquad\qquad V_t = \sum_{\tau=1}^{t} g_\tau^2$$

$$\eta_t = lr \cdot m_t / \left(\sqrt{V_t}\right)$$

$$= lr \cdot g_t / \left(\sqrt{\textstyle\sum_{\tau=1}^{t} g_\tau^2}\right)$$

$$w_{t+1} = w_t - \eta_t$$

$$= w_t - lr \cdot g_t / \left(\sqrt{\textstyle\sum_{\tau=1}^{t} g_\tau^2}\right)$$

优化器

✓Adagrad

一阶动量 当前时刻的梯度

$$m_t = g_t$$

二阶动量 梯度平方的累计和

$$V_t = \sum_{\tau=1}^{t} g_\tau^2$$

```
v_w, v_b = 0, 0
```

0时刻w和b的二阶动量初始值是0

```
# adagrad
v_w += tf.square(grads[0])
v_b += tf.square(grads[1])
w1.assign_sub(lr * grads[0] / tf.sqrt(v_w))
b1.assign_sub(lr * grads[1] / tf.sqrt(v_b))
```

源码：p36_adagrad.py

优化器

✓**RMSProp，SGD基础上增加二阶动量**

使用指数滑动平均值计算，表征过去一段时间的平均值

$$m_{\mathrm{t}} = g_t \qquad V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot g_t^2$$

$$\eta_t = lr \cdot m_{\mathrm{t}} / \sqrt{V_{\mathrm{t}}}$$

$$= lr \cdot g_t / (\sqrt{\beta \cdot V_{t-1} + (1-\beta) \cdot g_t^2})$$

$$w_{t+1} = w_t - \eta_t$$

$$= w_t - lr \cdot g_t / (\sqrt{\beta \cdot V_{t-1} + (1-\beta) \cdot g_t^2})$$

优化器

✓RMSProp

各时刻梯度平方的指数滑动平均

$$m_t = g_t \qquad V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot g_t^2$$

```
v_w, v_b = 0, 0
beta = 0.9
```

0时刻w和b的二阶动量初始值

```
# adadelta
v_w = beta * v_w + (1 - beta) * tf.square(grads[0])
v_b = beta * v_b + (1 - beta) * tf.square(grads[1])
w1.assign_sub(lr * grads[0] / tf.sqrt(v_w))
b1.assign_sub(lr * grads[1] / tf.sqrt(v_b))
```

源码：p38_rmsprop.py

优化器

✓**Adam, 同时结合SGDM一阶动量和RMSProp二阶动量** 并在此基础上增加了两个修正项

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$$

修正一阶动量的偏差：$\widehat{m_t} = \frac{m_t}{1-{\beta_1}^t}$

$$V_t = \beta_2 \cdot V_{step-1} + (1-\beta_2) \cdot g_t^2$$

修正二阶动量的偏差：$\widehat{V_t} = \frac{V_t}{1-{\beta_2}^t}$

$$\eta_t = lr \cdot \widehat{m_t} / \sqrt{\widehat{V_t}}$$

$$= lr \cdot \frac{m_t}{1-{\beta_1}^t} \Big/ \sqrt{\frac{V_t}{1-{\beta_2}^t}}$$

$$w_{t+1} = w_t - \eta_t$$

$$= w_t - lr \cdot \frac{m_t}{1-{\beta_1}^t} \Big/ \sqrt{\frac{V_t}{1-{\beta_2}^t}}$$

优化器

✓**Adam**

```
m_w, m_b = 0, 0
v_w, v_b = 0, 0
beta1, beta2 = 0.9, 0.999
delta_w, delta_b = 0, 0
global_step = 0
```

$$\widehat{m_t} = \frac{m_t}{1-{\beta_1}^t}$$

$$\widehat{V_t} = \frac{V_t}{1-{\beta_2}^t}$$

```
#adam
m_w = beta1 * m_w + (1 - beta1) * grads[0]
m_b = beta1 * m_b + (1 - beta1) * grads[1]
v_w = beta2 * v_w + (1 - beta2) * tf.square(grads[0])
v_b = beta2 * v_b + (1 - beta2) * tf.square(grads[1])
```

训练开始到当前时刻所经历的总batch数

```
m_w_correction = m_w / (1 - tf.pow(beta1, int(global_step)))
m_b_correction = m_b / (1 - tf.pow(beta1, int(global_step)))
v_w_correction = v_w / (1 - tf.pow(beta2, int(global_step)))
v_b_correction = v_b / (1 - tf.pow(beta2, int(global_step)))

w1.assign_sub(lr * m_w_correction / tf.sqrt(v_w_correction))
b1.assign_sub(lr * m_b_correction / tf.sqrt(v_b_correction))
```

源码：p40_adam.py