# 深度学习笔记-神经网络简介



## 感知器

感知器是神经网络的基础构成组件，可以看做节点组合。

## 一个简单的直线数据分类示例

对于坐标轴为 (p,q)(p,q) 的点，标签 y，以及等式 $\hat{y} = step(w_1x_1 + w_2x_2 + b)$

给出的预测

- 如果点分类正确，则什么也不做。
- 如果点分类为正，但是标签为负，则分别减去 $\alpha p, \alpha q$和 $\alpha$ 至 $w_1, w_2$和 $b$
- 如果点分类为负，但是标签为正，则分别将 $\alpha p, \alpha q$ 和 $\alpha$加到 $w_1, w_2$和 $b$上。

```python
# perceptron.py

import numpy as np
# Setting the random seed, feel free to change it and see different solutions.
np.random.seed(42)

def stepFunction(t):
    if t >= 0:
        return 1
    return 0

def prediction(X, W, b):
    return stepFunction((np.matmul(X,W)+b)[0])

# TODO: Fill in the code below to implement the perceptron trick.
# The function should receive as inputs the data X, the labels y,
# the weights W (as an array), and the bias b,
# update the weights and bias W, b, according to the perceptron algorithm,
# and return W and b.

def perceptronStep(X, y, W, b, learn_rate = 0.01):
```

```python
    # Fill in code
    return W, b

# This function runs the perceptron algorithm repeatedly on the dataset,
# and returns a few of the boundary lines obtained in the iterations,
# for plotting purposes.
# Feel free to play with the learning rate and the num_epochs,
# and see your results plotted below.
def trainPerceptronAlgorithm(X, y, learn_rate = 0.01, num_epochs = 25):
    x_min, x_max = min(X.T[0]), max(X.T[0])
    y_min, y_max = min(X.T[1]), max(X.T[1])
    W = np.array(np.random.rand(2,1))
    b = np.random.rand(1)[0] + x_max
    # These are the solution lines that get plotted below.
    boundary_lines = []
    for i in range(num_epochs):
        # In each epoch, we apply the perceptron step.
        W, b = perceptronStep(X, y, W, b, learn_rate)
        boundary_lines.append((-W[0]/W[1], -b/W[1]))
    return boundary_lines
```

```
# data.csv

0.78051,-0.063669,1
0.28774,0.29139,1
0.40714,0.17878,1
0.2923,0.4217,1
0.50922,0.35256,1
0.27785,0.10802,1
0.27527,0.33223,1
0.43999,0.31245,1
0.33557,0.42984,1
0.23448,0.24986,1
0.0084492,0.13658,1
0.12419,0.33595,1
0.25644,0.42624,1
0.4591,0.40426,1
0.44547,0.45117,1
0.42218,0.20118,1
0.49563,0.21445,1
0.30848,0.24306,1
0.39707,0.44438,1
0.32945,0.39217,1
0.40739,0.40271,1
0.3106,0.50702,1
0.49638,0.45384,1
0.10073,0.32053,1
0.69907,0.37307,1
0.29767,0.69648,1
0.15099,0.57341,1
0.16427,0.27759,1
0.33259,0.055964,1
0.53741,0.28637,1
```

```
0.19503,0.36879,1
0.40278,0.035148,1
0.21296,0.55169,1
0.48447,0.56991,1
0.25476,0.34596,1
0.21726,0.28641,1
0.67078,0.46538,1
0.3815,0.4622,1
0.53838,0.32774,1
0.4849,0.26071,1
0.37095,0.38809,1
0.54527,0.63911,1
0.32149,0.12007,1
0.42216,0.61666,1
0.10194,0.060408,1
0.15254,0.2168,1
0.45558,0.43769,1
0.28488,0.52142,1
0.27633,0.21264,1
0.39748,0.31902,1
0.5533,1,0
0.44274,0.59205,0
0.85176,0.6612,0
0.60436,0.86605,0
0.68243,0.48301,0
1,0.76815,0
0.72989,0.8107,0
0.67377,0.77975,0
0.78761,0.58177,0
0.71442,0.7668,0
0.49379,0.54226,0
0.78974,0.74233,0
0.67905,0.60921,0
0.6642,0.72519,0
0.79396,0.56789,0
0.70758,0.76022,0
0.59421,0.61857,0
0.49364,0.56224,0
0.77707,0.35025,0
0.79785,0.76921,0
0.70876,0.96764,0
0.69176,0.60865,0
0.66408,0.92075,0
0.65973,0.66666,0
0.64574,0.56845,0
0.89639,0.7085,0
0.85476,0.63167,0
0.62091,0.80424,0
0.79057,0.56108,0
0.58935,0.71582,0
0.56846,0.7406,0
0.65912,0.71548,0

0.70938,0.74041,0
```

```
0.59154,0.62927,0
0.45829,0.4641,0
0.79982,0.74847,0
0.60974,0.54757,0
0.68127,0.86985,0
0.76694,0.64736,0
0.69048,0.83058,0
0.68122,0.96541,0
0.73229,0.64245,0
0.76145,0.60138,0
0.58985,0.86955,0
0.73145,0.74516,0
0.77029,0.7014,0
0.73156,0.71782,0
0.44556,0.57991,0
0.85275,0.85987,0
0.51912,0.62359,0

```

```python
# solution.py

def perceptronStep(X, y, W, b, learn_rate = 0.01):
    for i in range(len(X)):
        y_hat = prediction(X[i],W,b)
        if y[i]-y_hat == 1:
            W[0] += X[i][0]*learn_rate
            W[1] += X[i][1]*learn_rate
            b += learn_rate
        elif y[i]-y_hat == -1:
            W[0] -= X[i][0]*learn_rate
            W[1] -= X[i][1]*learn_rate
            b -= learn_rate
    return W, b

```

# 误差函数

误差函数（ERROR)可以告诉我们目前的状况有多差，与理想解决方案的差别有多大。

# 离散型到连续型的转化

梯度下降只能用于连续型函数。对于一些离散型数据，将激活函数由跃迁函数改为s函数。

# softmax函数

```python
# softmax.py

import numpy as np


# Write a function that takes as input a list of numbers, and returns
```

```python
# the list of values given by the softmax function.
def softmax(L):
    expL = np.exp(L)
    sumExpL = sum(expL)
    result = []
    for i in expL:
        result.append(i*1.0/sumExpL)
    return result

    # Note: The function np.divide can also be used here, as follows:
    # def softmax(L):
    #     expL(np.exp(L))
    #     return np.divide (expL, expL.sum())

```

## 最大似然法

如在点的分类问题中，将每个点分类正确的概率相乘，得到所有点都分类正确的概率。然后尽可能地增大这个概率。这叫做最大似然法。

## 交叉熵

对最大似然法得到的概率进行求负对数，然后相加。越好的模型求得的交叉熵越小。 交叉熵公式：

```python
import numpy as np
# Write a function that takes as input two lists Y, P,
# and returns the float corresponding to their cross-entropy.
def cross_entropy(Y, P):
    Y = np.float_(Y)
    P = np.float_(P)
    return -np.sum(Y * np.log(P) + (1 - Y) * np.log(1 - P))
```

交叉熵公式只要保证只加上实际发生事件的概率负对数。

## 梯度计算

s型函数的导数：$\sigma'(x) = \sigma(x)(1-\sigma(x))$

误差公式是：$E = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \ln(\hat{y_i}) + (1-y_i)\ln(1-\hat{y_i})\right)$

预测是 $\hat{y_i} = \sigma(Wx^{(i)} + b)$

我们的目标是计算 E,E, 在点 $x = (x_1, \ldots, x_n)$ 时的梯度（偏导数）

$\nabla E = \left(\frac{\partial}{\partial w_1}E, \cdots, \frac{\partial}{\partial w_n}E, \frac{\partial}{\partial b}E\right)$

为此，首先我们要计算 $\frac{\partial}{\partial w_j}\hat{y}$.

最后得：$\nabla E(W,b) = (y-\hat{y})(x_1, \ldots, x_n, 1)$.

梯度实际上是标量乘以点的坐标.

# 梯度下降实验

- Sigmoid activation function

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

- Output (prediction) formula

$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$

- Error function

$Error(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$

- The function that updates the weights

$w_i \longrightarrow w_i + \alpha(y - \hat{y}) x_i$

$b \longrightarrow b + \alpha(y - \hat{y})$

代码实现：

```python
# Implement the following functions
# Activation (sigmoid) function
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Output (prediction) formula
def output_formula(features, weights, bias):
    return sigmoid(np.dot(features, weights) + bias)

# Error (log-loss) formula
def error_formula(y, output):
    return - y*np.log(output) - (1 - y) * np.log(1-output)

# Gradient descent step
def update_weights(x, y, weights, bias, learnrate):
    output = output_formula(x, weights, bias)
    d_error = y - output
    weights += learnrate * d_error * x
    bias += learnrate * d_error
    return weights, bias
```