**神经网络简介**

# 什么是神经网络

历史上，科学家一直希望模拟人的大脑，造出可以思考的机器。人为什么能够思考？科学家发现，原因在于人体的神经网络。



1. 外部刺激通过神经末梢，转化为电信号，转导到神经细胞（又叫神经元）。2. 无数神经元构成神经中枢。3. 神经中枢综合各种信号，做出判断。4. 人体根据神经中枢的指令，对外部刺激做出反应。

既然思考的基础是神经元，如果能够" 人造神经元"（artificial neuron），就能组成人工神经网络，模拟思考。上个世纪六十年代，提出了最早的" 人造神经元" 模型，叫做" 感知器"（perceptron），直到今天还在用。



#  logistic回归

logistic回归名为回归实际上是一个分类算法。是尝试使用机器学习方法解决二分类问题的第一步，一般先尝试使用logistic回归解决二分类问题，如果有明显效果，再尝试使用其它的分类算法。

## logistic 回归的计算公式

2.2 手动实现一个logistic 回归

### sigmoid 函数

In [3]: **import math**

**def** basic\_sigmoid(x):

*"""*

*Compute sigmoid of x.*

*Arguments:*

*x -- A scalar*

*Return:*

*s -- sigmoid(x)*

*"""*

*### START CODE HERE ### ( 1 line of code)*

s = 1/(1 + math.exp(-x))

*### END CODE HERE ###*

### sigmoid函数的导数

因为算法中需要进行梯度下降，索引需要对争相传播公式进行求导。



```python

# GRADED FUNCTION: sigmoid\_derivative

def sigmoid\_derivative(x):

 """

 Compute the gradient (also called the slope or derivative) of the sigmoid function with respect to its input x.

 You can store the output of the sigmoid function into variables and then use it to calculate the gradient.

 Arguments:

 x -- A scalar or numpy array

 Return:

 ds -- Your computed gradient.

 """

 ### START CODE HERE ### (≈ 2 lines of code)

 s = sigmoid(x)

 ds = s \* (1 - s)

 ### END CODE HERE ###

 return ds

```

### 解决一个图像分类问题

logistic回归解决图像分类问题的一般框架.从下图中也可一看出logistic回归和神经网络之间的关系。logistic回归是一个非常简单的神经网络。它只有一个神经元。



logistic回归的关键步骤：

1. 初始化模型的所有参数

2. 通过梯度下降法学习参数

3. 用训练好的模型进行预测

4. 总结和分析学到的模型

```python

# 导入所需要的模块

import numpy as np

import matplotlib.pyplot as plt

import h5py

import scipy

from PIL import Image

from scipy import ndimage

%matplotlib inline

```

```python

# 加载数据。此数据集是图片的集合，通过机器学习方法来判断图片中是不是一只猫

# Loading the data (cat/non-cat)

train\_set\_x\_orig, train\_set\_y, test\_set\_x\_orig, test\_set\_y, classes = load\_dataset()

# Example of a picture

index = 5

plt.imshow(train\_set\_x\_orig[index])

print ("y = " + str(train\_set\_y[:, index]) +

 ", it's a '" + classes[np.squeeze(train\_set\_y[:, index])].decode("utf-8") +

 "' picture.")

```

y = [0], it's a 'non-cat' picture.



```python

# 训练一个模型

# GRADED FUNCTION: model

def model(X\_train, Y\_train, X\_test, Y\_test, num\_iterations = 2000, learning\_rate = 0.5, print\_cost = False):

 """

 Builds the logistic regression model by calling the function you've implemented previously

 Arguments:

 X\_train -- training set represented by a numpy array of shape (num\_px \* num\_px \* 3, m\_train)

 Y\_train -- training labels represented by a numpy array (vector) of shape (1, m\_train)

 X\_test -- test set represented by a numpy array of shape (num\_px \* num\_px \* 3, m\_test)

 Y\_test -- test labels represented by a numpy array (vector) of shape (1, m\_test)

 num\_iterations -- hyperparameter representing the number of iterations to optimize the parameters

 learning\_rate -- hyperparameter representing the learning rate used in the update rule of optimize()

 print\_cost -- Set to true to print the cost every 100 iterations

 Returns:

 d -- dictionary containing information about the model.

 """

 ### START CODE HERE ###

 # initialize parameters with zeros (≈ 1 line of code)

 w, b = initialize\_with\_zeros(X\_train.shape[0])

 # Gradient descent (≈ 1 line of code)

 parameters, grads, costs = optimize(w, b, X\_train, Y\_train, num\_iterations, learning\_rate, print\_cost)

 # Retrieve parameters w and b from dictionary "parameters"

 w = parameters["w"]

 b = parameters["b"]

 # Predict test/train set examples (≈ 2 lines of code)

 Y\_prediction\_test = predict(w, b, X\_test)

 Y\_prediction\_train = predict(w, b, X\_train)

 ### END CODE HERE ###

 # Print train/test Errors

 print("train accuracy: {} %".format(100 - np.mean(np.abs(Y\_prediction\_train - Y\_train)) \* 100))

 print("test accuracy: {} %".format(100 - np.mean(np.abs(Y\_prediction\_test - Y\_test)) \* 100))

 d = {"costs": costs,

 "Y\_prediction\_test": Y\_prediction\_test,

 "Y\_prediction\_train" : Y\_prediction\_train,

 "w" : w,

 "b" : b,

 "learning\_rate" : learning\_rate,

 "num\_iterations": num\_iterations}

 return d

```

```python

# 预测一张picture

# Example of a picture that was wrongly classified.

index = 1

plt.imshow(test\_set\_x[:,index].reshape((num\_px, num\_px, 3)))

print ("y = " + str(test\_set\_y[0,index]) +

 ", you predicted that it is a \"" +

 classes[int(d["Y\_prediction\_test"][0,index])].decode("utf-8") +

 "\" picture.")

```

y = 1, you predicted that it is a "cat" picture.



# 多层感知机

## 神经网络结构



## 对一个二维数据集的分类结果



# 新兴的神经网络

##  deep learning

2006年，Hinton利用预训练方法缓解了局部最优解问题，将隐含层推动到了7层，神经网络真正意义上有了“深度”，由此揭开了深度学习的热潮。这里的“深度”并没有固定的定义——在语音识别中4层网络就能够被认为是“较深的”，而在图像识别中20层以上的网络屡见不鲜。为了克服梯度消失，ReLU、maxout等传输函数代替了sigmoid，形成了如今DNN的基本形式。单从结构上来说，全连接的DNN和下图的多层感知机是没有任何区别的。



##  CNN

我们需要识别一幅彩色图像，这幅图像具有四个通道ARGB（透明度和红绿蓝，对应了四幅相同大小的图像），假设卷积核大小为100\*100，共使用100个卷积核w1到w100（从直觉来看，每个卷积核应该学习到不同的结构特征）。用w1在ARGB图像上进行卷积操作，可以得到隐含层的第一幅图像；这幅隐含层图像左上角第一个像素是四幅输入图像左上角100\*100区域内像素的加权求和，以此类推。同理，算上其他卷积核，隐含层对应100幅“图像”。每幅图像对是对原始图像中不同特征的响应。按照这样的结构继续传递下去。CNN中还有max-pooling等操作进一步提高鲁棒性。



##  RNN

全连接的DNN还存在着另一个问题——无法对时间序列上的变化进行建模。然而，样本出现的时间顺序对于自然语言处理、语音识别、手写体识别等应用非常重要。对了适应这种需求，就出现了题主所说的另一种神经网络结构——循环神经网络RNN。

RNN网络结构如下图所示：



##  LSTM

为了解决时间上的梯度消失，机器学习领域发展出了长短时记忆单元LSTM，通过门的开关实现时间上记忆功能，并防止梯度消失，一个LSTM单元长这个样子：



# 一个神经网络的例子

python 中的内置sklearn库中已经实现了多种的神经网络。不需要自己根据公式去实现ANN算法。

下面是一个手写体识别的列子

```python

from sklearn.neural\_network import MLPClassifier

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

# 测试集，画图对预测值和实际值进行比较

def test\_validate(x\_test, y\_test, y\_predict, classifier):

 x = range(len(y\_test))

 plt.plot(x, y\_test, "ro", markersize=5, zorder=3, label=u"true\_v")

 plt.plot(x, y\_predict, "go", markersize=8, zorder=2, label=u"predict\_v,$R$=%.3f" % classifier.score(x\_test, y\_test))

 plt.legend(loc="upper left")

 plt.xlabel("number")

 plt.ylabel("true?")

 plt.show()

# 神经网络数字分类

def multi\_class\_nn():

 digits = datasets.load\_digits()

 x = digits['data']

 y = digits['target']

 # 对数据的训练集进行标准化

 ss = StandardScaler()

 x\_regular = ss.fit\_transform(x)

 # 划分训练集与测试集

 x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_regular, y, test\_size=0.1)

 clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden\_layer\_sizes=(5,), random\_state=1)

 clf.fit(x\_train, y\_train)

 # 模型效果获取

 r = clf.score(x\_train, y\_train)

 print("R值(准确率):", r)

 # 预测

 y\_predict = clf.predict(x\_test) # 预测

 print(y\_predict)

 print(y\_test)

 # 绘制测试集结果验证

 test\_validate(x\_test=x\_test, y\_test=y\_test, y\_predict=y\_predict, classifier=clf)

multi\_class\_nn()

```

 R值(准确率): 0.9938157081014224

 [9 0 5 8 7 4 3 4 4 3 1 2 3 8 0 3 4 1 9 4 2 1 8 3 3 5 3 2 5 6 8 8 3 9 1 2 8

 8 3 9 7 8 1 1 2 3 7 9 2 1 8 9 5 0 0 8 6 8 3 5 8 5 1 9 7 5 8 6 7 6 7 3 7 9

 3 9 5 0 0 5 7 7 3 8 0 2 0 7 7 1 5 4 1 3 9 5 8 9 7 1 8 9 1 0 0 7 8 0 9 7 5

 4 7 4 2 3 1 6 8 1 2 0 8 1 1 0 2 6 2 4 6 7 0 1 7 8 8 4 2 9 7 0 6 5 3 5 3 2

 3 8 9 9 5 0 4 8 3 6 9 3 6 9 0 5 1 4 2 5 3 4 7 9 5 1 9 3 5 0 8 4]

 [7 0 5 8 7 4 3 4 4 3 1 2 3 8 0 3 4 1 9 4 2 1 1 3 3 5 3 2 5 6 8 8 3 9 1 2 6

 8 3 9 7 8 1 1 2 3 7 9 2 1 1 9 5 0 0 1 6 4 3 5 8 8 1 9 7 5 8 6 7 6 7 3 7 9

 3 7 5 0 0 5 7 7 3 8 0 2 0 7 7 1 5 4 1 3 4 0 8 9 7 1 8 3 1 0 0 7 8 0 9 7 9

 4 7 4 2 9 1 6 4 1 2 0 7 7 1 0 2 6 3 4 6 4 0 1 7 8 6 4 2 9 7 0 6 5 3 5 3 2

 3 8 9 3 5 0 4 7 3 6 0 3 6 9 0 5 1 4 2 7 3 4 7 9 5 7 9 3 5 0 7 4]



红点是测试集真实结果，绿点是预测结果，红框部分出现了红绿点不重合部分数据（表示预测不准确），看到正确率r=0.872，预测正确/总数。