# 任务8 第一个情感分析程序

## 任务描述

**内 容**：使用KNN算法实现第一个情感分析程序。

**学 时**：2

**知识点**：KNN算法原理、KNN算法实现情感分析、KNN做情感分析的缺点

**重点**：KNN算法原理、KNN算法实现情感分析、KNN做情感分析的缺点

**难点**：KNN算法原理、KNN算法实现情感分析

## 授课思路

上2个任务，我们学习了计算机视觉中的人脸识别技术。从本任务开始，同学们开始学习计算机视觉中的图像分类技术。

本次任务的目的是引导出第一个图像分类算法。虽然这个算法有很多缺点，但是这是同学们入门图像分类的重要基础。我们会以多个任务为过渡，最终引出图像分类的强大算法-卷积神经网络。现在就进入卷积神经网络太难，我们需要一步一个脚印，慢慢接触。

本次任务在任务实现中提供了数据集和完整的代码。同学们通过学习理论知识和代码来掌握这个算法。

在编写程序之前，同学们必须先上网查资料了解KNN算法的原理和使用场景。

**特别说明**：人的表情分为多种，有生气、反感、害怕、高兴、中性、伤心、惊奇。为了快速让同学们入门，本次任务及后面的任务只关心高兴和中性2种表情。同学们有时间可以把这7种表情都考虑进去。数据源已提供，在 **补充数据源/fer** 目录下。

## 任务指导

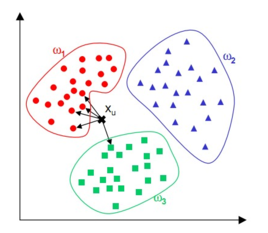
1. KNN算法的原理
2. 使用KNN实现情感分析

## 任务实现

1. KNN算法的原理

邻近算法，或者说K最近邻(kNN，k-NearestNeighbor)分类算法是[数据挖掘](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E6%8C%96%E6%8E%98/216477)分类技术中最简单的方法之一。所谓K最近邻，就是k个最近的邻居的意思，说的是每个样本都可以用它最接近的k个邻居来代表。

kNN算法的核心思想是如果一个样本在特征空间中的k个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。 kNN方法在类别决策时，只与极少量的相邻样本有关。由于kNN方法主要靠周围有限的邻近的样本，而不是靠判别类域的方法来确定所属类别的，因此对于类域的交叉或重叠较多的待分样本集来说，kNN方法较其他方法更为适合。



算法流程：

1）准备数据，对数据进行[预处理](https://baike.baidu.com/item/%E9%A2%84%E5%A4%84%E7%90%86)

2）选用合适的数据结构存储训练数据和测试元组

3）设定参数，如k

4）维护一个大小为k的的按距离由大到小的[优先级队列](https://baike.baidu.com/item/%E4%BC%98%E5%85%88%E7%BA%A7%E9%98%9F%E5%88%97)，用于存储最近邻训练元组。随机从训练元组中选取k个元组作为初始的最近邻元组，分别计算测试元组到这k个元组的距离，将训练元组标号和距离存入优先级队列

5）遍历训练元组集，计算当前训练元组与测试元组的距离，将所得距离L 与优先级队列中的最大距离Lmax

6）进行比较。若L>=Lmax，则舍弃该元组，遍历下一个元组。若L < Lmax，删除优先级队列中最大距离的元组，将当前训练元组存入优先级队列。

7）遍历完毕，计算优先级队列中k 个元组的多数类，并将其作为测试元组的类别。

8）测试元组集测试完毕后计算误差率，继续设定不同的k值重新进行训练，最后取误差率最小的k 值。

参考资料：

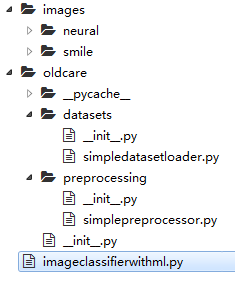
<https://baike.baidu.com/item/%E9%82%BB%E8%BF%91%E7%AE%97%E6%B3%95/1151153?fr=aladdin>

<https://www.jianshu.com/p/3dcb39de04aa>

1. 使用KNN实现情感分析

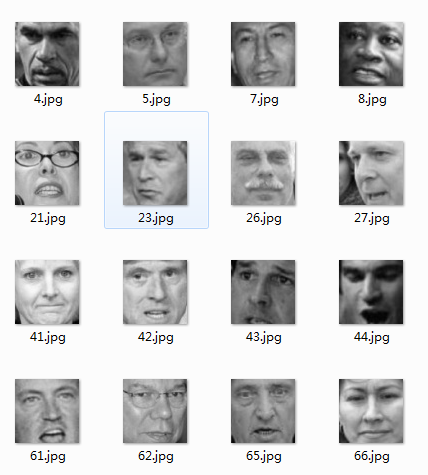
本次任务完整代码存放在 **任务源代码/任务8.第一个情感分析程序** 中。

任务层级结构如下图所示：

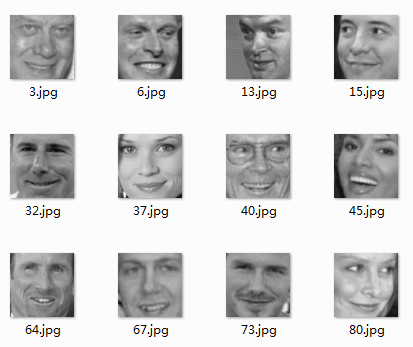


images目录下存放着表情数据集。images目录下有2个子目录，分别叫neural和smile。其中neural目录下存放着所有没有笑的图片，smile目录下存放着所有笑的图片。

neural目录下的文件如下图所示：



smile目录下存放的图片如下图所示：



在根目录下新建一个package，命名为**oldcare**。

在oldcare这个package下新建一个package，命名为**datasets**。

在datasets这个package下新建一个module，命名为**simpledatasetloader.py**。此文件用于载入数据。

打开\_\_init\_\_.py，添加1行代码。代码如下：

from .simpledatasetloader import SimpleDatasetLoader

打开simpledatasetloader.py，添加如下代码：

# -\*- coding: utf-8 -\*-

# import the necessary packages

import numpy as np

import cv2

import os

class SimpleDatasetLoader:

def \_\_init\_\_(self, preprocessors=None):

# store the image preprocessor

self.preprocessors = preprocessors

# if the preprocessors are None, initialize them as an

# empty list

if self.preprocessors is None:

self.preprocessors = []

def load(self, imagePaths, verbose = 500, grayscale = False):

# initialize the list of features and labels

data = []

labels = []

# loop over the input images

for (i, imagePath) in enumerate(imagePaths):

# load the image and extract the class label assuming

# that our path has the following format:

# /path/to/dataset/{class}/{image}.jpg

image = cv2.imread(imagePath)

if grayscale:

image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

label = imagePath.split(os.path.sep)[-2]

# check to see if our preprocessors are not None

if self.preprocessors is not None:

# loop over the preprocessors and apply each to

# the image

for p in self.preprocessors:

image = p.preprocess(image)

# treat our processed image as a "feature vector"

# by updating the data list followed by the labels

data.append(image)

labels.append(label)

# show an update every `verbose` images

if verbose > 0 and i > 0 and (i + 1) % verbose == 0:

print("[INFO] processed {}/{}".format(i + 1,

len(imagePaths)))

# return a tuple of the data and labels

return (np.array(data), np.array(labels))

在oldcare这个package下新建一个package，命名为**preprocessing**。

在prprocessing这个package下面新建一个module，命名为**simplepreprocessor.py**。这个文件的作用是在载入图片的同时，对图片进行处理。最基本的处理就是改变图片尺寸。

打开\_\_init\_\_.py，添加1行代码。代码如下：

from .simplepreprocessor import SimplePreprocessor

打开simplepreprocessor.py中，添加如下代码：

# -\*- coding: utf-8 -\*-

'''

process data

'''

import cv2

class SimplePreprocessor:

def \_\_init\_\_(self, width, height, inter=cv2.INTER\_AREA):

# store the target image width, height, and interpolation

#method used when resizing

self.width = width

self.height = height

self.inter = inter

def preprocess(self, image):

# resize the image to a fixed size, ignoring the aspect ratio

return cv2.resize(image, (self.width, self.height),

interpolation=self.inter)

在根目录下新建一个module，命名为**imageclassifierwithml.py**。

打开imageclassifierwithml.py文件，添加代码如下：

# -\*- coding: utf-8 -\*-

'''

image classification with knn

'''

from oldcare.preprocessing import SimplePreprocessor

from oldcare.datasets import SimpleDatasetLoader

from imutils import paths

from sklearn.preprocessing import LabelEncoder

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import classification\_report

################################################3

# 第一部分：数据预处理

# 全局变量

dataset\_path = 'images'

# 全局常量

N\_NEIGHBOURS = 5

TARGET\_IMAGE\_WIDTH = 32

TARGET\_IMAGE\_HEIGHT = 32

# initialize the image preprocessor and datasetloader

sp = SimplePreprocessor(TARGET\_IMAGE\_WIDTH, TARGET\_IMAGE\_HEIGHT)

sdl = SimpleDatasetLoader(preprocessors=[sp])

# Load images

print("[INFO] loading images...")

image\_paths = list(paths.list\_images(dataset\_path)) # path included

(X, y) = sdl.load(image\_paths, verbose=500, grayscale = True)

# Flatten (reshape the data matrix)

# convert from (13164,32,32) into (13164,32\*32)

X = X.reshape((X.shape[0], TARGET\_IMAGE\_WIDTH\*TARGET\_IMAGE\_HEIGHT))

# Show some information on memory consumption of the images

print("[INFO] features matrix: {:.1f}MB"

.format(X.nbytes / (1024 \* 1024.0)))

# Label encoder

le = LabelEncoder()

y = le.fit\_transform(y)

# Split dataset

(X\_train, X\_test, y\_train, y\_test) = train\_test\_split(X, y,

test\_size=0.25,

random\_state=42)

################################################3

# 第二部分：训练模型

# Train model

print("[INFO] evaluating k-NN classifier...")

model = KNeighborsClassifier(n\_neighbors= N\_NEIGHBOURS,

metric = 'minkowski', p = 2)

model.fit(X\_train, y\_train)

################################################

# 第三部分：评估模型

# Evaluate model

y\_pred = model.predict(X\_test)

# Confusion Matrix

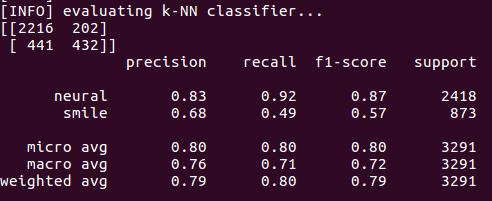
cm = confusion\_matrix(y\_test, y\_pred)

print(cm)

# Report

print(classification\_report(y\_test, y\_pred, target\_names=le.classes\_))

执行imageclassifierwithml.py，得到模型的准确性为79%，如下图。效果看起来还不错。



KNN虽然在本例中效果不错，但是KNN用于图像分类整体上准确率还是比较低的。同学们有时间测试一下彩色图像的分类。补充数据源/dogs\_cats 中提供了彩色猫和狗的共2000张图片。

这是用机器学习的一种方法实现图像分类。从下个任务开始，将详细探讨深度学习。

# 任务9人工神经网络（ANN）认知-1

## 任务描述

**内 容**：理解并掌握ANN的原理。

**学 时**：2

**知识点**：单层感知器架构、多层感知器架构、激活函数

**重点**：单层感知器架构、多层感知器架构、激活函数

**难点**：激活函数

## 授课思路

上个任务，我们使用机器学习的一种方法实现了图像分类。从本任务开始，我们接触深度学习，并最终使用深度学习的方法对图像进行分类。

使用Python实现深度学习的代码量是非常少的。对于深度学习，代码是非常简单的，难的是理解深度学习的原理，并能够优化模型。

本任务让同学们专注于理解人工神经网络的原理。人工神经网络是深度学习的重要基础。本任务的实现方式是：教师先大体解释一下某个概念，然后学生查阅资料，一段时间后师生互动加深理解这个概念。

本次任务对学生而言是一个比较大的挑战。

## 任务指导

1. 什么是深度学习
2. 深度学习发展历程
3. 单层感知器架构
4. 多层感知器架构
5. 激活函数

## 任务实现

1. 什么是深度学习



深度学习之父：Geoffrey Hinton

**深度学习主要模拟了人脑对新事物的反应、例如学习行为，它粗略地构建了人脑中神经以及神经突触的层级关系。**

人脑有大量被称为神经元的脑细胞，人之所以能够进行思维，就是因为这些神经元互相联结，处理各种信息。

同样，如果用机器模拟神经元，并建立人工神经元联结网络，这样的人工神经网络就能够进行思维。



机器学习就是让计算机自动学习相关知识并解决实际问题。

**实现机器学习有很多方法，深度学习即深度神经网络是其中重要的方法之一。**

参考资料：

<https://blog.csdn.net/antony1776/article/details/74637248>

<http://www.sohu.com/a/198034375_99970711>

<https://blog.csdn.net/antony1776/article/details/74637248>

[http://playground.tensorflow.org](http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.21139&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&co)

1. 深度学习发展历程

1943年，神经科学家和控制论专家Warren McCulloch和逻辑学家Walter Pitts基于数学和阈值逻辑算法创造了一种神经网络计算模型；

1957年，心理学家Frank Rosenblatt创造了模式识别算法感知机，用简单的加减算法实现了两层的计算机学习网络；

1974年，Paul Werbos在博士论文中提出了用误差反向传导来训练人工神经网络有效解决了异或回路问题，使得训练多层神经网络称为可能；

1985年，Rumelhart和McClelland提出了BP网络误差反向传播学习算法；

1998年，以Yann Lecun为首的研究人员实现了一个七层的卷积神经网络LeNet-5识别手写数字；

2006年，以Geoffrey Hinton为代表的加拿大高等研究院附属机构的研究人员开始将人工神经网络/联结主义重新包装为深度学习并进行推广；

2009-2012年，瑞士人工智能实验室IDSIA发展了递归神经网络和深前馈神经网络；

2012年，Geoffrey Hinton组的研究人员在ImageNet2012上夺冠，他们的图像分类效果远远超过了第二名，深度学习热潮由此开始一直持续到现在；

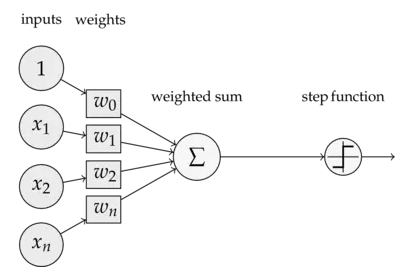
参考资料：

<http://mini.eastday.com/mobile/180314030343551.html>

<https://baijiahao.baidu.com/s?id=1587853812420778645&wfr=spider&for=pc>

1. 单层感知器架构

单层感知器由一个线性组合器和一个二值阈值元件组成。

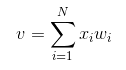


输入向量为x，权重向量为w，w0为偏执。

简单的理解可以解释为：将x0,x1······xn的变量输入，经过组合器的整合，输出1或者0，也就是通过组合器对输入变量判断其正确与否。

而这个判断的依据就是权重w0,w1······wn。

因为线性组合器是实现加法的方式，根据向量的运算法则，所以以上公式的输入值可以理解为：  
w0+x1w1+······+xnwn



参考资料：

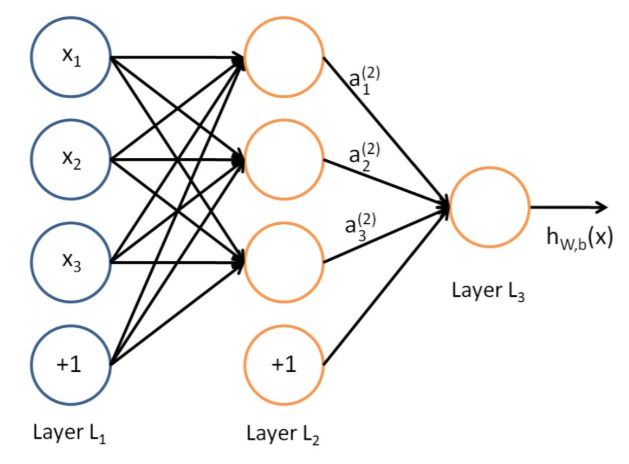
<https://www.jianshu.com/p/d7189cbd0983?from=groupmessage>

<http://neuralnetworksanddeeplearning.com/chap1.html>

1. 多层感知器架构

多层感知器(Multi-Layer Perceptron，MLP)也叫人工神经网络(Artificial Neural Network，ANN)，除了输入输出层，它中间可以有多个隐层。最简单的MLP需要有一层隐层，即输入层、隐层和输出层才能称为一个简单的神经网络。习惯原因我之后会称为神经网络。通俗而言，神经网络是仿生物神经网络而来的一种技术，通过连接多个特征值，经过线性和非线性的组合，最终达到一个目标，这个目标可以是识别这个图片是不是一只猫，是不是一条狗或者属于哪个分布。

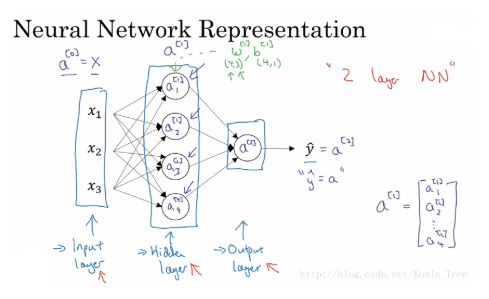
首先来看一个简单MLP的模型：



在这个模型中，我们输入的x特征会连接到隐层的神经元，隐层的神经元再连接到输入层的神经元。在这个多层感知器层与层之间是全连接的(全连接的意思就是：上一层的任何一个神经元与下一层的所有神经元都有连接)。

神经网络要解决的最基本问题是分类问题。我们将特征值传入隐层中，通过带有结果的数据来训练神经网络的参数(W，权重；b，偏置)，使输出值与我们给出的结果一致，既可以用来预测新的输入值了。

接下来，我们给出一个单隐层的神经网络，以此来看一下神经网络的具体流程。大致模型如图所示：



首先我们先对图中的参数做一个解释：

其中，第0层(输入层)，我们将x1，x2和x3向量化为X；

0层和1层(隐层)之间，存在权重w1(x1到各个隐层)，w2...w4，向量化为W[1]，其中[1]表示第1层的权重，偏置b同理；

对于第1层，计算公式为：

Z[1] = W[1]X + b[1]

A[1] = sigmoid(Z[1])

其中Z为输入值的线性组合，A为Z通过激活函数sigmoid的值，对于第1层的输入值为X，输出值为A，也是下一层的输入值；

1层和2层(输出层)之间，与0层和1层之间类似，其计算公式如下：

Z[2] = W[2]A[1] + b[2]

A[2] = sigmoid(Z[2])

yhat = A[2]

参考资料：

<https://blog.csdn.net/weixin_38206214/article/details/81137911>

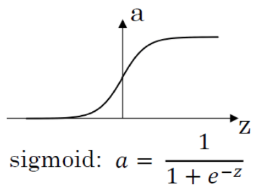
<http://neuralnetworksanddeeplearning.com/chap1.html>

<https://cs.stanford.edu/people/karpathy/convnetjs//demo/classify2d.html>

1. 激活函数

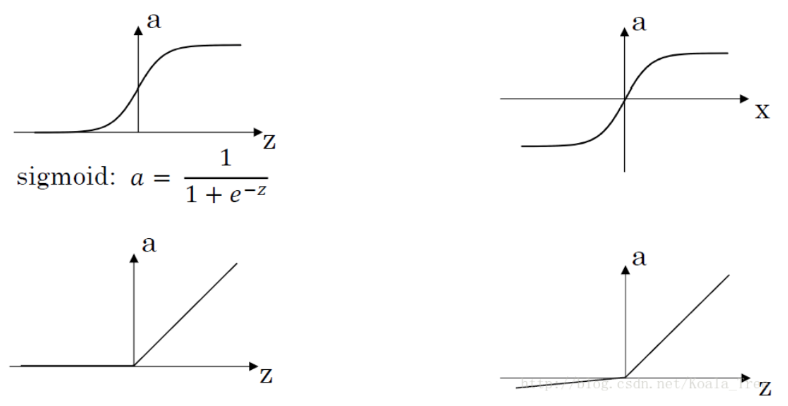
为什么要用激活函数

根据输出的计算可以发现，其实隐层的每个神经元是由输入特征x的线性组合构成。然而如果仅仅是线性组合，那么不管这个神经网络有多少层，结果都将与特征线性相关。于是我们在每个神经元结果z之后，添加一个激活函数(Activation Function)，改变线性规则，比如使用Sigmoid函数，公式如下：



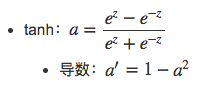
在Sigmoid函数中，a的值在[0,1]之间，我们可以将其理解为一个阀。就像是人的神经元一样，当我们一个神经元受到的刺激时，我们并不是立刻感觉到，而是当这个刺激值超过了阀值，才会让神经元向上级神经元发出信号。

接下里我们给出常用的几个激活函数以及其图像：



图上左上角，即为上文所述的sigmoid函数，其导数为a· = a(1 - a)

右上角为tanh函数：



左下角为ReLU(修正线性单元)：a = max(0，z)

右下角为Leaky ReLU：a = max(0.01，z)

对于激活函数的选择，是神经网络中很重要的一步，对比sigmoid函数和tanh函数在不同层的使用效果：

隐藏层：tanh函数的表现要好于sigmoid函数，因为tanh的取值范围在[-1，1]之间，均值为0，实际上气到了归一化(使图像分布在0周围，得到的结果更方便使用梯度下降)的效果。

输出层：对于二分类而言，实际上是在计算yhat的概率，在[0，1]之间，所以sigmoid函数更优。

然而在sigmoid函数和ReLU函数中，当Z很大或很小时，Z的导数会变得很小，趋紧于0，这也称为梯度消失，会影响神经网络的训练效率。

ReLU函数弥补了二者的缺陷，当z>0时，梯度始终为1，从而提高神经网络的运算速度。然而当z<0时，梯度始终为0。但在实际应用中，该缺陷影响不是很大。

Leaky ReLU是对ReLU的补偿，在z<0时，保证梯度不为0。

总结而言

在选择激活函数的时候，如果不知道该选什么的时候就选择ReLU，当然具体训练中，需要我们去探索哪种函数更适合。

参考资料：

https://blog.csdn.net/weixin\_38206214/article/details/81137911

<https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>

<https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks>

# 任务10 人工神经网络（ANN）认知-2

## 任务描述

**内 容**：理解并掌握ANN的原理。

**学 时**：2

**知识点**：反向传播算法

**重点**：反向传播算法

**难点**：反向传播算法

## 授课思路

本次任务承接上次任务继续学习ANN的原理。

如果上个任务没有完成，本次任务可以继续上个任务的内容。

本次任务主要学习深度学习最重要的算法：反向传播算法。有了这个算法深度学习才真正发展起来。请同学们仔细学习。

## 任务指导

1. 反向传播算法

## 任务实现

1. 反向传播算法

**Background**

Backpropagation is a common method for training a neural network. There is [no shortage of papers](https://www.google.com/search?q=backpropagation+algorithm) online that attempt to explain how backpropagation works, but few that include an example with actual numbers. This post is my attempt to explain how it works with a concrete example that folks can compare their own calculations to in order to ensure they understand backpropagation correctly.

If this kind of thing interests you, you should [sign up for my newsletter](http://www.emergentmind.com/newsletter) where I post about AI-related projects that I’m working on.

**Backpropagation in Python**

You can play around with a Python script that I wrote that implements the backpropagation algorithm in [this Github repo](https://github.com/mattm/simple-neural-network).

**Backpropagation Visualization**

For an interactive visualization showing a neural network as it learns, check out my [Neural Network visualization](http://www.emergentmind.com/neural-network).

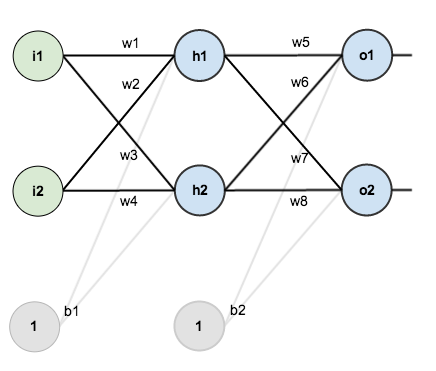
**Additional Resources**

If you find this tutorial useful and want to continue learning about neural networks, machine learning, and deep learning, I highly recommend checking out Adrian Rosebrock’s new book, [Deep Learning for Computer Vision with Python](https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/). I really enjoyed the book and will have a full review up soon.

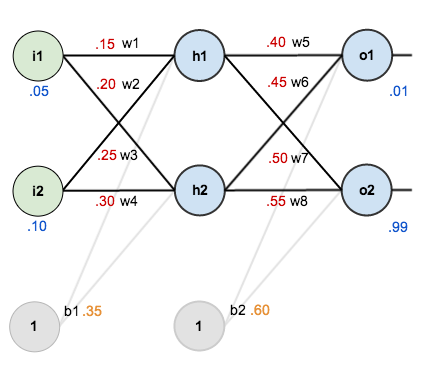
**Overview**

For this tutorial, we’re going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

Here’s the basic structure:



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs:



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

For the rest of this tutorial we’re going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

**The Forward Pass**

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we’ll feed those inputs forward though the network.

We figure out the total net input to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons.

Total net input is also referred to as just *net input* by [some sources](http://web.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf).

Here’s how we calculate the total net input for h_1:

net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1

net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775

We then squash it using the logistic function to get the output of h_1:

out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992

Carrying out the same process for h_2 we get:

out_{h2} = 0.596884378

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here’s the output for o_1:

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967

out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507

And carrying out the same process for o_2 we get:

out_{o2} = 0.772928465

**Calculating the Total Error**

We can now calculate the error for each output neuron using the [squared error function](http://en.wikipedia.org/wiki/Backpropagation#Derivation) and sum them to get the total error:

E_{total} = \sum \frac{1}{2}(target - output)^{2}

[Some sources](http://www.amazon.com/Introduction-Math-Neural-Networks-Heaton-ebook/dp/B00845UQL6/ref=sr_1_1?ie=UTF8&qid=1426296804&sr=8-1&keywords=neural+network) refer to the target as the *ideal* and the output as the *actual*.

The \frac{1}{2} is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway so it doesn’t matter that we introduce a constant here [[1](http://en.wikipedia.org/wiki/Backpropagation#Derivation)].

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^{2} = \frac{1}{2}(0.01 - 0.75136507)^{2} = 0.274811083

Repeating this process for o_2 (remembering that the target is 0.99) we get:

E_{o2} = 0.023560026

The total error for the neural network is the sum of these errors:

E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109

**The Backwards Pass**

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

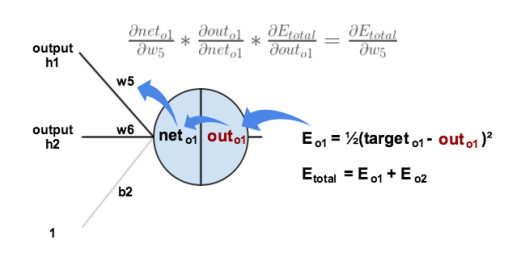
**Output Layer**

Consider w_5. We want to know how much a change in w_5 affects the total error, aka \frac{\partial E_{total}}{\partial w_{5}}.

\frac{\partial E_{total}}{\partial w_{5}} is read as “the partial derivative of E_{total} with respect to w_{5}“. You can also say “the gradient with respect to w_{5}“.

By applying the [chain rule](http://en.wikipedia.org/wiki/Chain_rule) we know that:

Visually, here’s what we’re doing:



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^{2} + \frac{1}{2}(target_{o2} - out_{o2})^{2}

\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2 - 1} * -1 + 0

\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507

-(target - out) is sometimes expressed as out - target

When we take the partial derivative of the total error with respect to out_{o1}, the quantity \frac{1}{2}(target_{o2} - out_{o2})^{2} becomes zero because out_{o1} does not affect it which means we’re taking the derivative of a constant which is zero.

Next, how much does the output of o_1 change with respect to its total net input?

The partial [derivative of the logistic function](http://en.wikipedia.org/wiki/Logistic_function#Derivative) is the output multiplied by 1 minus the output:

out_{o1} = \frac{1}{1+e^{-net_{o1}}}

\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602

Finally, how much does the total net input of o1 change with respect to w_5?

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

\frac{\partial net_{o1}}{\partial w_{5}} = 1 * out_{h1} * w_5^{(1 - 1)} + 0 + 0 = out_{h1} = 0.593269992

Putting it all together:

\frac{\partial E_{total}}{\partial w_{5}} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041

You’ll often see this calculation combined in the form of the [delta rule](http://en.wikipedia.org/wiki/Delta_rule):

\frac{\partial E_{total}}{\partial w_{5}} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}

Alternatively, we have \frac{\partial E_{total}}{\partial out_{o1}} and \frac{\partial out_{o1}}{\partial net_{o1}} which can be written as \frac{\partial E_{total}}{\partial net_{o1}}, aka \delta_{o1} (the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})

Therefore:

\frac{\partial E_{total}}{\partial w_{5}} = \delta_{o1} out_{h1}

Some sources extract the negative sign from \delta so it would be written as:

\frac{\partial E_{total}}{\partial w_{5}} = -\delta_{o1} out_{h1}

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we’ll set to 0.5):

w_5^{+} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_{5}} = 0.4 - 0.5 * 0.082167041 = 0.35891648

[Some](http://en.wikipedia.org/wiki/Delta_rule) [sources](http://aima.cs.berkeley.edu/) use \alpha (alpha) to represent the learning rate, [others use](https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf) \eta(eta), and [others](http://web.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf) even use \epsilon (epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

w_6^{+} = 0.408666186

w_7^{+} = 0.511301270

w_8^{+} = 0.561370121

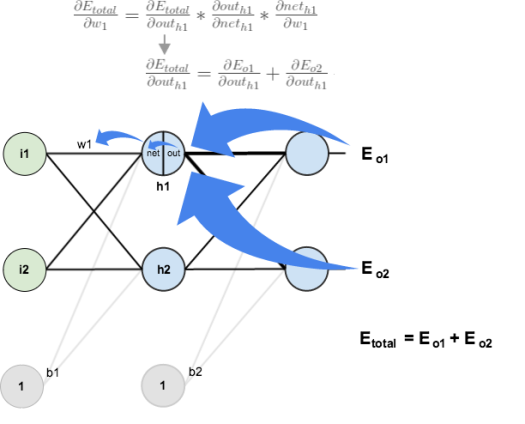
We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

**Hidden Layer**

Next, we’ll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here’s what we need to figure out:

Visually:

[](https://matthewmazur.files.wordpress.com/2015/03/nn-calculation.png)

We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the \frac{\partial E_{total}}{\partial out_{h1}} needs to take into consideration its effect on the both output neurons:

Starting with \frac{\partial E_{o1}}{\partial out_{h1}}:

We can calculate \frac{\partial E_{o1}}{\partial net_{o1}} using values we calculated earlier:

And \frac{\partial net_{o1}}{\partial out_{h1}} is equal to w_5:

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40

Plugging them in:

Following the same process for \frac{\partial E_{o2}}{\partial out_{h1}}, we get:

\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119

Therefore:

Now that we have \frac{\partial E_{total}}{\partial out_{h1}}, we need to figure out \frac{\partial out_{h1}}{\partial net_{h1}} and then \frac{\partial net_{h1}}{\partial w} for each weight:

out_{h1} = \frac{1}{1+e^{-net_{h1}}}

\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999 ) = 0.241300709

We calculate the partial derivative of the total net input to h_1 with respect to w_1the same as we did for the output neuron:

net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1

\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05

Putting it all together:

\frac{\partial E_{total}}{\partial w_{1}} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568

You might also see this written as:

\frac{\partial E_{total}}{\partial w_{1}} = (\sum\limits_{o}{\delta_{o} * w_{ho}}) * out_{h1}(1 - out_{h1}) * i_{1}

\frac{\partial E_{total}}{\partial w_{1}} = \delta_{h1}i_{1}

We can now update w_1:

w_1^{+} = w_1 - \eta * \frac{\partial E_{total}}{\partial w_{1}} = 0.15 - 0.5 * 0.000438568 = 0.149780716

Repeating this for w_2, w_3, and w_4

w_2^{+} = 0.19956143

w_3^{+} = 0.24975114

w_4^{+} = 0.29950229

Finally, we’ve updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

If you’ve made it this far and found any errors in any of the above or can think of any ways to make it clearer for future readers, don’t hesitate to [drop me a note](https://mattmazur.com/contact/). Thanks!

参考资料：

<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

# 任务11 使用人工神经网络（ANN）做情感分析

## 任务描述

**内 容**：使用ANN做情感分析。

**学 时**：2

**知识点**：ANN的具体编程实现、超参数、ANN的优化

**重点**：ANN的具体编程实现、超参数、ANN的优化

**难点**：ANN的优化

## 授课思路

上个任务，我们学习了ANN的原理。本任务对ANN做一次实践。

这是同学们的第一个深度学习程序，任务实现中会详细贴上所有的代码。同学们通过看代码加深对ANN的理解。

## 任务指导

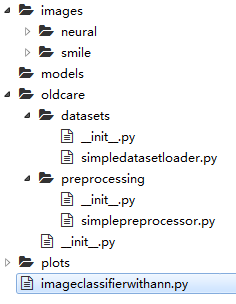
1. ANN的具体编程实现
2. ANN的优化

## 任务实现

本次任务完整代码存放在 **任务源代码/任务11.使用人工神经网络(ANN)做情感分析** 中。

1. ANN的具体编程实现

本次任务的层级结构如下图：



其中images目录大家已经很熟悉了，这里不再赘述。

oldcare里面的内容大家已经很熟悉了，这里不再赘述。

models目录我们也已经很熟悉了，为了把训练好的模型保存到里面。

plots目录是新添加的目录，作用是存放模型训练的曲线。这条曲线用于评估模型的好坏。

在根目录中新建一个module，命名为**imageclassifierwithann.py**。在imageclassifierwithann.py中添加如下代码：

# -\*- coding: utf-8 -\*-

'''

使用ANN做情感分析

'''

# 导入包

from oldcare.preprocessing import SimplePreprocessor

from oldcare.datasets import SimpleDatasetLoader

from imutils import paths

from sklearn.preprocessing import LabelEncoder

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

from sklearn.metrics import classification\_report

from keras.models import Sequential

from keras.layers.core import Dense

from keras.optimizers import SGD

from keras.utils import to\_categorical

import numpy as np

import matplotlib.pyplot as plt

# 全局变量

dataset\_path = 'images'

accuracy\_plot\_path = 'plots/accuracy.png'

loss\_plot\_path = 'plots/loss.png'

output\_model\_path = 'models/face\_expression.hdf5'

# 全局常量

TARGET\_IMAGE\_WIDTH = 28

TARGET\_IMAGE\_HEIGHT = 28

LR = 0.001 # 学习率

BATCH\_SIZE = 64

EPOCHS = 40

################################################

# 第一部分：数据预处理

# initialize the image preprocessor and datasetloader

sp = SimplePreprocessor(TARGET\_IMAGE\_WIDTH, TARGET\_IMAGE\_HEIGHT)

sdl = SimpleDatasetLoader(preprocessors=[sp])

# Load images

print("[INFO] 导入图像...")

image\_paths = list(paths.list\_images(dataset\_path)) # path included

(X, y) = sdl.load(image\_paths, verbose=500, grayscale = True)

# Flatten (reshape the data matrix)

# convert from (13164,TARGET\_IMAGE\_WIDTH,TARGET\_IMAGE\_HEIGHT)

#into (13164,TARGET\_IMAGE\_WIDTH\*TARGET\_IMAGE\_HEIGHT)

X = X.reshape((X.shape[0], TARGET\_IMAGE\_WIDTH\*TARGET\_IMAGE\_HEIGHT))

X = X.astype("float") / 255.0 # 特征缩放，是非常重要的步骤

# Show some information on memory consumption of the images

print("[INFO] features matrix: {:.1f}MB"

.format(X.nbytes / (1024 \* 1024.0)))

# Label encoder

le = LabelEncoder()

y = to\_categorical(le.fit\_transform(y), 2)

print(le.classes\_)

# 拆分数据集

(X\_train, X\_test, y\_train, y\_test) = train\_test\_split(X, y,

test\_size=0.25,

random\_state=42)

################################################3

# 第二部分：创建并训练模型

# 创建模型

model = Sequential()

model.add(Dense(1024,

input\_shape=(TARGET\_IMAGE\_WIDTH \* TARGET\_IMAGE\_HEIGHT,),

activation="relu"))

model.add(Dense(512, activation="relu"))

model.add(Dense(2, activation="softmax"))

# 训练模型

print("[INFO] 训练模型...")

sgd = SGD(LR)

model.compile(loss="categorical\_crossentropy", optimizer=sgd,

metrics=["accuracy"])

H = model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test),

epochs=EPOCHS, batch\_size=BATCH\_SIZE, verbose=1)

################################################

# 第三部分：评估模型

# 画出accuracy曲线

plt.style.use("ggplot")

plt.figure()

plt.plot(np.arange(1, EPOCHS+1), H.history["acc"], label="train\_acc")

plt.plot(np.arange(1, EPOCHS+1), H.history["val\_acc"],label="val\_acc")

plt.title("Training Accuracy")

plt.xlabel("Epoch #")

plt.ylabel("Accuracy")

plt.legend()

plt.savefig(accuracy\_plot\_path)

# 画出loss曲线

plt.style.use("ggplot")

plt.figure()

plt.plot(np.arange(1, EPOCHS+1),H.history["loss"],label="train\_loss")

plt.plot(np.arange(1,EPOCHS+1),H.history["val\_loss"],label="val\_loss")

plt.title("Training Loss")

plt.xlabel("Epoch #")

plt.ylabel("Loss")

plt.legend()

plt.savefig(loss\_plot\_path)

# 打印分类报告

label\_names = le.classes\_.tolist()

print("[INFO] 评估模型...")

predictions = model.predict(X\_test, batch\_size=BATCH\_SIZE)

print(classification\_report(y\_test.argmax(axis=1),

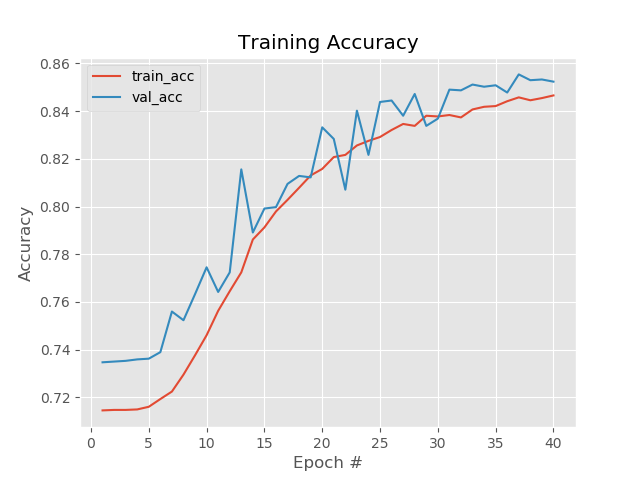
predictions.argmax(axis=1), target\_names=label\_names))

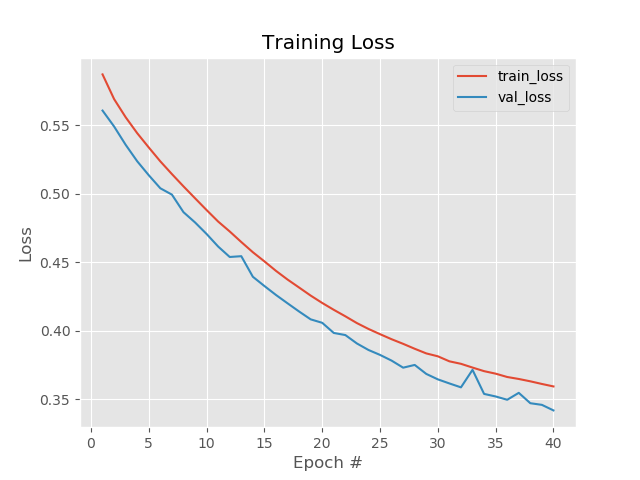
################################################

# 第四部分：保存模型

model.save(output\_model\_path)

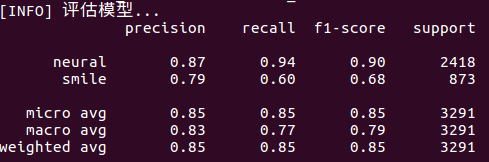
执行imageclassifierwithann.py会打印出模型训练过程中的**accuray**和**loss**曲线，如下图：





这两条曲线具有**非常大**的意义，从中能看出许多信息。

同第一个情感分析程序一样，我们打印出模型的评估报告如下图：



可以看出，准确率较KNN算法有较大的提高。

1. ANN的优化

请同学们修改 **全局常量** 中5个常量的值，然后重新训练模型。比较loss和accuracy曲线有哪些变化，并思考为什么。