第5章 神经网络算法基础

本章学习目标

- 了解神经网络的基本概念；
- 掌握神经网络中激活函数的作用和用法；
- 通过 TensorFlow 实现单层神经网络；
- 了解通过 TensorFlow 实现神经网络常见层的方法。

神经网络算法在图像识别、语音识别、自动驾驶等领域的应用越来越深入，并取得了优秀的成果。深度学习的发展建立在单层神经网络的基础之上，它是一类通过多层非线性变换对高复杂性数据建模的算法的集合，由于深层神经网络常常被用于实现多层非线性变换，从某种程度上来说，“深度学习”可以被看作“深层神经网络”的代名词。本章介绍神经网络算法以及在 TensorFlow 中实现较为简单的神经网络算法的方法。

5.1 神经网络算法简介

人工神经网络是由大量具有适应性的神经元组成的广泛并行互联网络，它的组织能够模拟生物神经系统对真实世界物体所做出的的交互反应，是模拟人工智能的重要途径之一。通常提到的“神经网络”，实际上是指“神经网络学习”。人工神经网络的学习方法中的连接主义通过编写一个初始模型，然后通过数据训练，不断改善模型中的参数，直到输出的结果符合预期，便实现了“学习”。在网络层次上模拟人的思维过程中的某些神经元的层级组合，用人脑的并行处理模式，来表征认知过程。这种受神经科学启发的机器学习方法，被称为人工神经网络方法。

神经网络解决问题的步骤如下所示。

(1) 提取实际问题中数据的特征向量作为神经网络的输入数据。这意味着解决问题时，首先要对数据集实施特征工程，求得每个样本的特征维度，根据样本特征的维度来定义输入神经元的数量。

(2) 构建神经网络，定义神经网络中由输入得到输出的过程，即定义神经网络的输入层、隐藏层和输出层。

(3) 通过迭代训练数据来优化神经网络中的参数。这一过程往往需要通过定义模型的损失函数和参数优化方法来实现，如常见的交叉熵损失函数和梯度下降算法等。

(4) 评估训练好的模型，用训练好的模型预测未知的数据，通常根据预测的准确率来衡量模型的性能。

5.2 TensorFlow 实现激活函数

激活函数的主要作用是调节权重和误差。例如，在数字识别的神经网络中，图像“1”被误分为“7”时，通过细微的调整权重和偏差值对模型进行优化，使网络更准确地分类形状接近“1”的图像。但是，在一个网络的众多神经元当中，任何参数的细微调整，都可能会使输出发生巨大的变化。因此可能出现图像“1”虽然被正确分类，但是网络中的神经元无法学习到为其他数字的图像进行分类的“规则”，这也就使得网络中参数学习变得极其困难。

在神经网络中添加激活函数为其引入了非线性因素，使得权重和偏差的变化不再对输出产生过大的影响，以此达到微调网络的目的。这样神经网络就可以应用到众多的非线性模型中，更好地解决较为复杂的问题。如果没有激活函数，每一层输出都是上层输入的线性函数，无论神经网络有多少层，输出都是输入的线性组合。在 TensorFlow 中，激活函数是作用于张量上的非线性操作，本节将介绍几种常见的激活函数。

5.2.1 Sigmoid 函数

需要注意的是，判断激活函数是否适合所应用模型的主要标准为，该激活函数是否可以让优化整个深度神经网络的过程更加便捷。Sigmoid 函数是一种常见的激活函数，其工作原理如图 5.1 所示。

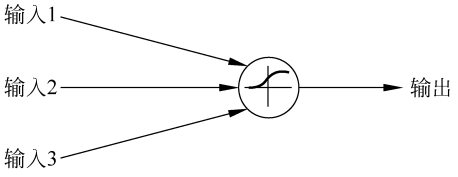

图 5.1 Sigmoid 神经元

通过微调权值(weight)和偏差(bias)让神经网络的输出结果产生相应的轻微改变。

Sigmoid 函数的输出区间为(0,1)，输入区间为$(-\infty,+\infty)$，该函数的表达式如下所示。

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

Sigmoid 函数的图像如图 5.2 所示。

Sigmoid 函数全程可导，它比阶跃函数更加平滑，阶跃函数如图 5.3 所示。

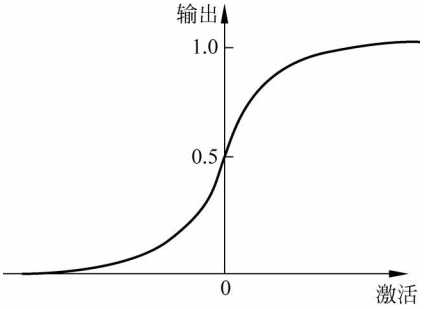

图 5.2 Sigmoid 函数

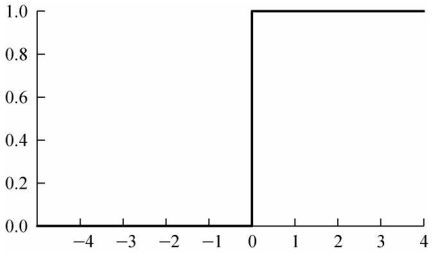

图 5.3 阶跃函数

Sigmoid 函数的在 Tensorflow 中的实现代码如下所示。

```
import tensorflow as tf
input_data = tf.Variable( [[0, 10, -10],[1,2,3]] , dtype = tf.float32 )
output = tf.nn.sigmoid(input_data)
with tf.Session() as sess:
    init = tf.initialize_all_variables()
    sess.run(init)
    print(sess.run(output))
```

输出如下所示。

```
[[ 5.00000000e-01 9.99954581e-01 4.53978719e-05]
 [ 7.31058598e-01 8.80797029e-01 9.52574134e-01]]
```

然而，近年来 Sigmoid 函数由于其某些突出的缺点导致使用率不断下降，主要缺点如下所示。

(1) 当出现极大或极小的输入数据时，Sigmoid 函数在输出接近 0 或 1 的区域时会饱和，函数在饱和区域的梯度变化非常平缓，接近于 0，这很容易造成梯度消失的问题。

(2) Sigmoid 函数的输出不是 0 均值的，这很可能导致在学习过程中，上一层的神经元的非 0 均值输出被作为输入传递给下一层神经元，在不断迭代中很容易放大误差。

(3) 使用 Sigmoid 函数在处理含有大量数据的模型时收敛速度较慢，而深度学习往往需要处理海量的数据。

5.2.2 Tanh 函数

Tanh 函数是双曲正切函数，属于 Sigmoid 函数的变形，它是对 Sigmoid 函数的逼近或者近似，对 Sigmoid 因离散性而导致的难以优化的缺陷的弥补。与 Sigmoid 函数不同的是，Tanh 函数是 0 均值的，它的输出以 0 为中心，Tanh 函数将整个实数区间的输入值映射到区间(−1,1)中。因此，在输入数据的特征差异明显时，Tanh 函数会在循环过程中不断扩大特征差异，这种情况下 Tanh 函数具有比 Sigmoid 函数更好的收敛效果。由于 Tanh 函数存在软饱和性，因此 Tanh 函数依然存在梯度消失的问题。

Tanh 函数的数学公式为：

$$\mathrm{Tanh}(x)=\frac{\sinh(x)}{\cosh(x)}$$

在函数取值范围(−1,1)时函数图像如图 5.4 所示

Tanh 函数公式中 $\sinh(x)$数学表达式为：

$$\sinh(x)=\frac{e^{x}-e^{-x}}{2}$$

$\cosh(x)$的表达式为：

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

Tanh 函数与 Sigmoid 函数的差别：

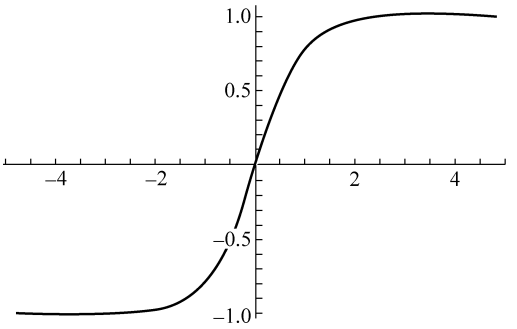

图 5.4　Tanh 函数

(1) 由于 Tanh 函数的导数值域为(0,1],而 Sigmoid 函数的导数值域为(0,0.25],可以看出 Tanh 函数的导数值域是大于 Sigmoid 函数的导数值域的,因此在反向传播的过程中,Tanh 函数比 Sigmoid 函数延迟了饱和期。

(2) 通过观察 Tanh 函数与 Sigmoid 函数的函数曲线可以看出,Tanh 函数的输入和输出能够保持非线性单调上升和下降关系,符合反向传播算法的梯度求解,容错性好,有界,渐近于 0、1,符合人脑神经饱和的规律。而 Sigmoid 函数在输入处于[-1,1]之间时,函数值变化敏感,一旦接近或者超出该段区间函数便会失去敏感性,趋于饱和状态,影响神经网络。

5.2.3　ReLU 数

ReLU 函数是目前最为常见的激活函数之一,全称为 Rectified Linear Units,称为线性修正单元。该函数的表达式为 $y=\max(0,x)$,当 $x>0$时 $f'(x)=1$,当 $x\leqslant 0$ 时,$f'(x)=0$。ReLU 函数可以看作是一个求最大值函数,具体如图 5.5 所示。

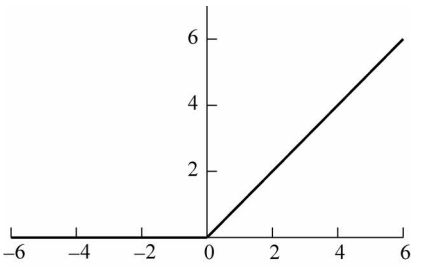

图 5.5　ReLU 函数

从图 5.5 中不难看出,ReLU 函数不是全区间可导的,它具有以下几点优势。

(1) 在正区间上解决了梯度消失问题。

(2) 单侧抑制,从图 5.5 中可以看到,在输入小于或等于 0 时,输出为 0,这表明神经元此时处于抑制状态;当输入大于 0 时,神经元处于激活状态。这使得 ReLU 函数只需要判断输出是否大于 0,因此极大地提高其了计算效率。

(3) ReLU 函数会将抑制状态的神经元置 0,这些被置 0 的神经元不会参与后续的计算,这大大提升了收敛速度,收敛速度快于 Sigmoid 函数和 Tanh 函数。

ReLU 函数的实现方法如下所示。

(1) 首先导入相关工具库,并创建会话。

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

(2) 生成一个 ReLU 函数。

```
import tensorflow as tf
sess = tf.InteractiveSession()
# ReLU 函数处理负数
print("anwser 1:",tf.nn.relu(-2.9).eval())
# ReLU 函数处理正数
print("anwser 2:",tf.nn.relu(3.4).eval())
# 产生一个 4×4 的矩阵,满足均值为 0,标准差为 1 的正态分布
a = tf.Variable(tf.random_normal([4,4],mean=0.0, stddev=1.0))
# 对所有变量进行初始化,这里对 a 进行初始化
tf.global_variables_initializer().run()
# 输出原始的 a 的值
print("原始矩阵:\n",a.eval())
# 对 a 使用 Relu 函数进行激活处理,将结果保存到 b 中
b = tf.nn.relu(a)
# 输出处理后的 a,即 b 的值
print("ReLU 函数激活后的矩阵:\n",b.eval())
```

输出如下所示。

```
anwser 1: 0.0
anwser 2: 3.4
原始矩阵:
  [[-1.259318   -0.09126675   -2.454077    -1.376778 ]
  [-0.61337525   1.91146       0.92145157   0.45843443]
  [-1.6872858   -0.5119289     1.416437     1.146053 ]
  [  0.22473626 -0.03396741    0.7743713    0.7691514 ]]
ReLU 函数激活后的矩阵:
  [[0.           0.           0.           0.         ]
  [0.           1.91146      0.92145157   0.45843443]
  [0.           0.           1.416437     1.146053 ]
  [0.22473626   0.           0.7743713    0.7691514 ]]
```

需要注意的是,ReLU 函数的输出不是 0 均值化的,并且某些神经元会因为 ReLU 函数的置 0 操作导致其永远不会被激活,这使得相应的参数永远无法被更新。

5.3 TensorFlow 实现单层神经网络

本章前两节对神经网络算法的基础应用方法进行了基本的介绍,接下来将通过一个较为具体的单层神经网络来进一步讲解全连接神经网络算法的矩阵乘法运算等内容。

以 iris 数据集为基础,通过单层神经网络来解决回归算法问题,在解决回归算法问题时,建议以均方误差作为损失函数。具体代码如下所示。

(1) 加载必要的工具包,创建计算图会话。

```
import matplotlib.pyplot as plt
import numpy as np
```

```
import tensorflow as tf
from sklearn import datasets
```

(2) 定义用于训练神经网络的训练数据集。定义一个范围在[－1,1]之间的等差数列数据 x_data,包含 300 个数据,以及 x_data 平方后加上偏差－0.5 和噪声的 y_data,这是训练神经网络用的原始数据集。

```
x_data = np.linspace( - 1,1,300)[:,np.newaxis]
noise = np.random.normal(0,0.05,xdata.shape)
y_data = np.square(xdata) - 0.5 + noise
```

(3) 神经网络结构的定义可以分层实现,先分别定义各层,然后再组合起来,下面是一个定义神经网络层的函数。

```
def addlayer(inputdata,input_size,out_size,active = None):
    weights = tf.Variable(tf.random_normal([input_size,out_size]))
    bias = tf.Variable(tf.zeros([1,out_size]) + 0.1)
    wx_plus_b = tf.matmul(inputdata,weights) + bias
    if active == None:
        return wx_plus_b
    else:
        return active(wx_plus_b)
```

代码中 weights 表示权重值。前一层每个输出节点同本层所有节点构成的权值矩阵大小为 input_szie×out_size,即本层权值链接大小为 input_size×out_size。上述代码通过声明 weights 构造了一个初始值个数为 input_size×out_size,且权值符合标准正态分布的权值矩阵。

同理,每层的神经元个数对应于该层的输出个数,每个输出都必须有一个对应的偏差值,因此上述代码通过声明 bias,定义了初始值为 0.1 的偏差值,偏差值的个数等于 out_size。

在定义了权重和偏差以后,通过声明 wx_plus_b 来实现神经元的信息接收和偏差值的计算。最后判断是否设置了激活函数,如果已设置,则在返回中输出激活后的结果,否则,直接返回未被激活的输出。

(4) 定义输入数据。

```
xinput = tf.placeholder(tf.float32,[None,1])
youtput = tf.placeholder(tf.float32,[None,1])
```

(5) 接下来,定义一个单输入,单输出,具有单隐层(节点个数为 10)的神经的 3 层神经网络(含输入层):

(6) 定义神经网络的损失函数为均方误差损失函数。

```
loss = tf.reduce_mean(tf.reduce_sum(tf.square(youtput - output),reduction_indices = [1]))
```

(7) 定义训练目标。

```
train = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
```

通常用 Tensorflow 中默认的梯度下降的方法来最小化损失函数作为训练目标。

(8) 初始化全局变量。

```
init = tf.initialize_all_variables()
```

(9) 迭代训练神经网络。

```
with tf.Session() as sess:
    sess.run(init) #进行初始化变量
    for i in range(2000):
        sess.run(train,feed_dict = {xinput:xdata,youtput:ydata})
        if i % 100 == 0:
            print(i,sess.run(loss,feed_dict = {xinput:xdata,youtput:ydata}))
```

通过 Session 激活相应的操作,设置迭代次数为 2000 次,每 100 次迭代打印一次训练参数。输出如下所示。

```
0     0.683216
100   0.0152671
200   0.00892168
300   0.00744904
400   0.00715826
500   0.00674129
600   0.00639426
700   0.00612397
800   0.00572691
900   0.00564182
1000  0.00549631
1100  0.00536184
1200  0.00518188
1300  0.00506481
1400  0.00498346
1500  0.00484763
1600  0.00474545
1700  0.00465182
1800  0.00458013
1900  0.00453197
```

5.4 TensorFlow 实现神经网络常见层

5.3 节介绍了实现单层神经网络的方法,本节将在 5.3 节所学知识的基础上加入实现包括神经网络的卷积层和池化层的方法。有关卷积操作和池化操作的具体内容后将在续章

节详细讲解。此处列出，只作为了解。

(1) 首先，加载相关工具库并创建计算图会话。

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
```

(2) 初始化数据，这里输入数据规格大小为 10×10。

```
data_size = [10,10]
data_2d = np.random.normal(size = data_size)
x_input_2d = tf.placeholder(tf.float32,shape = data_size)
```

(3) 声明卷积层函数，这里采用 TensorFlow 内建函数 tf.nn.conv2d()，由于该函数需要输入 4 维数据(批量大小，宽度，高度，颜色通道)，因此首先需要对输入数据进行扩维。tf.expand.dims 函数为扩维函数，其作用是在给定位置增加一维。例如输入数据 shape=[10,10]，则经过 tf.expand_dims(input_2d,0)操作后变成了[1,10,10]。

其次配置相应的滤波器、步长值以及填充值。滤波器选用外部输入滤波器参数 myfilter，两个方向的步长值均为 2，设置填充值的参数为 VALID。采用 tf.squeeze()函数将卷积后的数据规格转换为二维。

```
def conv_layer_2d(input_2d , myfilter):
    # 将数据转化为 4 维
    input_3d = tf.expand_dims(input_2d,0) #shape = [1,10,10]
    input_4d = tf.expand_dims(input_3d,3) #shape = [1,10,10,1]
    # 卷积操作
    # 这里设置两个方向上的补偿,padding 选择 no padding,myfileter 采用 2 * 2,
    # 因此 10 * 10 ==> 5 * 5
    conv_out = tf.nn.conv2d(input = input_4d,filter = myfilter,
               strides = [1,2,2,1],padding = 'VALID')
    #维度还原
    conv_out_2d = tf.squeeze(conv_out)
    return conv_out_2d
```

(4) 定义卷积核的大小，经过上述创建的卷积层后，输出数据 shape 为[5,5]。

```
myfilter = tf.Variable(tf.random_normal(shape = [2,2,1,1]))
my_convolution_output = conv_layer_2d(x_input_2d,myfilter)
```

(5) 声明激活函数，激活函数是针对逐个元素的，创建激活函数并初始化后将上述卷积层得到的数据通过激活函数激活。

```
def activation(input_2d):
    return tf.nn.relu(input_2d)
my_activation_output = activation(my_convolution_output)
```

(6) 声明一个池化层,输入为经激活函数后的数据,shape=[5,5]。池化层采用 TensorFlow 内建函数,其处理方式与卷积层类似,需要先扩维,然后通过 tf.nn.max_pool()函数进行池化操作,最后降维得到输出数据,这里池化层步长为 1,宽和高为 2×2,shape 变为[4,4]。

```
def max_pool(input_2d,width,height):
    #先将数据扩展为 4 维
    input_3d = tf.expand_dims(input_2d,0) #shape = [1,5,5]
    input_4d = tf.expand_dims(input_3d,3) #shape = [1,5,5,1]
    #池化操作
    pool_output = tf.nn.max_pool(input_4d,ksize=[1,height,width,1],
                  strides=[1,1,1,1],padding='VALID')
    #降维
    pool_output_2d = tf.squeeze(pool_output)#shape = [4,4]
    return pool_output_2d
my_maxpool_output = max_pool(my_activation_output,width=2,height=2)
```

(7) 设置全连接层,输入为经池化层后的数据,shape=[4,4]以及需要连接的神经元个数 num_outputs=5。为了保证所有数据均能够与神经元相连接,需要先将数据转化为一维向量,并计算 $y=wx+b$ 中权重值 w 和偏差值 b 的规格。输入数据的规格变为[16],通过 tf.shape()函数得到相应的 shape 规格,然后通过 tf.stack()函数将输入 shape 与神经元个数连接得到规格为[[16],[5]],最后通过 tf.squeeze()函数得到权重值 w 的 shape 为[16,5],偏差值 b 的 shape 为 5。最后由于 $y=wx+b$ 为矩阵运算,需要将数据扩维至二维数据,经计算后再通过降维还原回一维数据,这里输出为 5 个神经元的值。

```
def fully_connected(input_layer,num_outputs):
    #首先将数据转化为一维向量,以实现每项连接到每个输出
    flat_input = tf.reshape(input_layer,[-1])
    #创建 w 和 b
    #确定 w 和 b 的 shape
    #tf.shape 得到数据的大小,例如 4×4 的数据变为向量,则 shape=16,
    #tf.stack 为矩阵拼接,设 num_outputs = 5,则其结果为[[16],[5]]
    #tf.squeeze 降维,使其结果为[16,5]满足 shape 输入格式要求
    weight_shape = tf.squeeze(tf.stack([tf.shape(flat_input),[num_outputs]]))
    weight = tf.random_normal(shape=weight_shape,stddev=0.1)
    bias = tf.random_normal(shape=[num_outputs])
    #将数据转化为二维以完成矩阵乘法
    input_2d = tf.expand_dims(flat_input,0)
    #进行计算 y=wx+b
    fully_output = tf.add(tf.matmul(input_2d,weight),bias)
    #降维
    fully_output_result = tf.squeeze(fully_output)
    return fully_output_result
my_full_output = fully_connected(my_maxpool_output,num_outputs=5)
```

(8) 初始化全局变量并写入计算图中,然后输出结果。

```
#初始化变量
init = tf.initialize_all_variables()
```

```
sess.run(init)
feed_dict = {x_input_2d:data_2d}
#打印各层
#卷积层
print('Input = [10 10] array')
print('Convolution [2,2],stride size = [2,2],
      results in the [5,5] array:')
print(sess.run(my_convolution_output,feed_dict = feed_dict))
#激活函数输出
print('Input = the above [5,5] array ')
print('ReLU element wise returns the [5,5] array ')
print(sess.run(my_activation_output,feed_dict = feed_dict))
#池化层输出
print('Input = the above [5,5] array ')
print('Maxpool,stride size = [1,1], results in the [4,4] array ')
print(sess.run(my_maxpool_output,feed_dict = feed_dict))
#全连接层输出
print('Input = the above [4,4] array ')
print('Fully connected layer on all four rows with five outputs:')
print(sess.run(my_full_output,feed_dict = feed_dict))
```

输出结果如下所示。

```
Input = [10 10] array
Convolution [2,2],stride size = [2,2], results in the [5,5] array:
[[ 1.32886136     - 1.47333026     - 1.44128537       - 0.95050871     - 1.80972886]
[ - 2.82501674    - 0.35346282     - 0.06931959         1.9739815      - 0.84173405]
[   0.5519557     - 1.66942024       0.56509626       - 2.68546128       0.71953934]
[ - 3.13675737    - 1.81401241       1.47897935       - 0.1665355        0.05618015]
[   2.81271505    - 4.40996552     - 1.39324057         1.17697966     - 2.26855183]]
Input = the above [5,5] array
ReLU element wise returns the [5,5] array
[[1.32886136    0.    0.            0.          0.          ]
[ 0.            0.    0.            1.9739815   0.          ]
[ 0.5519557     0.    0.56509626    0.          0.71953934]
[ 0.            0.    1.47897935    0.          0.05618015]
[ 2.81271505    0.    0.            1.17697966  0.          ]]
Input = the above [5,5] array
Maxpool,stride size = [1,1], results in the [4,4] array
[[1.32886136    0.              1.9739815     1.9739815 ]
[ 0.5519557     0.56509626      1.9739815     1.9739815 ]
[ 0.5519557     1.47897935      1.47897935    0.71953934]
[ 2.81271505    1.47897935      1.47897935    1.17697966]]
Input = the above [4,4] array
Fully connected layer on all four rows with five outputs:
[ - 1.14044487    0.18718313    2.26356006    - 0.60274446    0.6560365 ]
```

5.5 本章小结

通过本章的学习，大家可以初步掌握神经网络算法的相关基础，理解激活函数的意义和作用，并掌握其使用方法。掌握实现简单的单层神经网络的方法和实现其他常见网络结构的方法，为接下来的深入学习打好基础。

5.6 习　　题

1. 填空题

(1) 激活函数的主要作用是调节_____和_____，它为神经网络引入了非线性因素。

(2) Sigmoid 函数的表达式为_____。

(3) TensorFlow 的常见激活函数中，Tanh 函数为 0 均值函数，它的输出以_____为中心。

2. 选择题

(1) 下列选项中属于 ReLU 函数的特点是(　　)。

A. 单侧抑制　　B. 全区间可导

C. 0 均值　　D. 输出区间为(0,1)

(2) Sigmoid 函数的输出区间为(　　)，输入区间为(　　)。

A. (0,1),(0,+∞)　　B. (−1,1),(−∞,+∞)

C. (0,1),(−∞,+∞)　　D. (−1,1),(0,+∞)

(3) 下列激活函数中，在正区间上解决了梯度消失问题的函数是(　　)。

A. Sigmoid 函数　　B. Tanh 函数

C. ReLU 函数　　D. 以上选项皆错

3. 思考题

简述激活函数在神经网络模型构建中的意义。